TURING 图灵程序设计丛书

PACKT PUBLISHING

React Design Patterns and Best Practices

React设计模式与最佳实践

[意] 米凯莱·贝尔托利 著

林昊 译

- Facebook前端工程师15年一线开发经验凝结
- 深入探讨React核心模式与组件，创建可复用的代码和可扩展的设计

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

作者简介

米凯莱·贝尔托利
（Michele Bertoli）

Facebook前端工程师，曾任职于YPlan和BIZZBY等公司，拥有超过15年的实践经验。他喜欢整洁且经过充分测试的代码，目前致力于使用React.js来开发现代JavaScript应用。

封面设计 | 83720326@qq.com
自在书装设计

数字版权声明

图灵程序设计丛书

React Design Patterns and Best Practices

React设计模式与最佳实践

[意] 米凯莱·贝尔托利 著
林昊 译

人民邮电出版社
北京

图书在版编目（C I P）数据

React设计模式与最佳实践 / (意) 米凯莱·贝尔托利 (Michele Bertoli) 著 ; 林昊 译. -- 北京 : 人民邮电出版社, 2018.8
（图灵程序设计丛书）
ISBN 978-7-115-48875-6

Ⅰ. ①R… Ⅱ. ①米… ②林… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2018)第157759号

内 容 提 要

本书共分为 12 章，通过介绍 React 中最有价值的设计模式，展示如何将设计模式和最佳实践应用于现实的新项目和已有项目中。主要内容包括帮助读者理解 React 的基本概念，学习编写整洁、可维护的代码；优化 React 组件，使应用拥有更快的速度和响应性；介绍如何有效地编写测试，避免反模式，开源组件并对 React 生态系统做贡献。

本书适合想要深入理解 React，希望提高相关编程技能的前端开发人员阅读。

◆ 著　　　　[意] 米凯莱·贝尔托利
　译　　　　林　昊
　责任编辑　杨　琳
　责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　印刷
◆ 开本：800×1000　1/16
　印张：14.75
　字数：357千字　　　　2018年 8 月第 1 版
　印数：1－3 000册　　　2018年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2017-5047号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

致　　谢

我要感谢我的妻子和儿子，他们用微笑让我的生活变得更美好。Dante，我将一些时间用来撰写这本书而没有陪你玩耍，希望等你长大后能理解并因为我而感到骄傲。另外，还要感谢Packt出版社给予我这次机会。

前　　言

本书将带你全面了解 React 中最有价值的设计模式，并展示如何在全新或已有的真实项目中应用设计模式与最佳实践。本书将帮助你让应用变得更加灵活、运行更流畅并且更容易维护——在不降低质量的情况下极大地提升工作流的速度。

我们将首先理解 React 的内部原理，接着逐步编写整洁且可维护的代码。我们将开发能够在整个应用中复用的组件，搭建应用架构，并创建真正可用的表单。

接下来，我们会为 React 组件编写样式并优化组件，从而使应用运行得更快且更具响应性。最后，我们将有效地编写测试代码，还会学到如何为 React 及其生态系统做贡献。

学完本书后，你会从大量的试错以及开发难题中解脱出来，也将踏上成为 React 专家的道路。

本书内容

第 1 章，React 基础。这一章从高级角度介绍了 React 的基本概念。

第 2 章，整理代码。这一章讲解了编写可维护代码中最重要的一个方面，即保持代码整洁并遵循编程风格指南。了解函数式编程的基础知识对于使用 React 也很重要。

第 3 章，开发真正可复用的组件。这一章阐述了构建应用的一个关键因素在于使用组件，而要想保持代码库整洁且可维护，最重要的是开发真正可复用的组件。

第 4 章，组合一切。这一章阐述了真实应用由不同的组件构成，重要的是让组件之间可以高效地通信，并按照正确的形式组织和搭建层次结构。

第 5 章，恰当地获取数据。这一章指明了任何客户端应用在某些时刻都必须处理数据，并且介绍了不同的技巧和方法，让你能够以 React 的方式获取数据。

第 6 章，为浏览器编写代码。这一章阐述了如何正确使用在浏览器中运行的应用，还讲解了一些高级概念，如事件、动画以及如何与 DOM 交互。

第 7 章，美化组件。这一章说明了开发美观的 UI 组件是前端工程中很重要的一部分内容。

React 可以通过多种方式实现这个目的，每种方式从不同角度解决该问题。了解可用的库及其工作原理，对于做出正确的选择至关重要。

第 8 章，服务端渲染的乐趣与益处。这一章指明了服务端渲染是 React 众多优秀特性之一。虽然该特性开箱即用，但学习其正确用法很重要，因为这样才能充分加以利用。

第 9 章，提升应用性能。这一章阐述了性能是 Web 平台吸引用户的重要因素之一。React 提供了一系列工具和技术来创建快如闪电的应用，这一章将全面介绍这些内容。

第 10 章，测试与调试。这一章会让你意识到，我们都希望自己的应用保持稳定，并且能够应对一切极端情况，而测试有助于实现这个目的。编写全面的测试集对于创建稳定且可维护的代码至关重要。从另一方面来看，bug 总会出现，而知道如何调试并尽早发现问题很关键。

第 11 章，需要避免的反模式。这一章阐明了开发人员经常尝试采取捷径和创意方案这一事实，但在某些情况下这种做法对应用来说是很危险的，尤其是团队以及代码库规模很大时。这一章将带你了解使用 React 时，应该避免的常见反模式。

第 12 章，未来的行动。这是本书的最后一章，至此我们已经介绍完所有主题。我认为探讨如何开源组件（以回馈社区）以及如何为 React 及其生态系统做贡献也很重要。

阅读须知

我们需要一台计算机，并配有终端程序、Node.js/npm 环境以及浏览器。

目标读者

如果想要深入理解 React 并将其应用到真实应用的开发中，那么本书很适合你。

排版约定

本书采用不同的文本样式来区分不同类别的信息。以下展示了部分样式示例及其相应的含义。

正文中的代码、数据库表名、用户输入等采用以下样式：“循环内部包含一些条件逻辑，用于检查`#first` 和`#link` 属性是否存在，并根据它们的值渲染不同的 HTML 片段。变量位于双花括号中。”

代码块的样式如下所示：

```
const toLowerCase = input => {
  const output = []
  for (let i = 0; i < input.length; i++) {
    output.push(input[i].toLowerCase())
  }
  return output
}
```

命令行中的输入或输出内容采用以下样式：

```
npm install -g create-react-app
```

新术语和**关键词**以黑体字显示。屏幕上出现的单词（如出现在菜单或对话框中）按照如下样式显示："我们开始更新测试代码，先从**渲染文本**的那些代码着手。"

警告或重要的注意事项。

提示或小技巧。

读者反馈

我们期待读者的反馈。告诉我们你对本书的看法，喜欢什么或者不喜欢什么。读者反馈对我们很重要，因为它有助于我们策划出令读者受益最多的图书。

要想提供反馈，只需登录"图灵社区"本书页面（http://www.ituring.com.cn/book/2007）并留言。

客户支持

为了让你购买的书物有所值，我们还为你准备了以下内容。

下载示例代码

你可以从"图灵社区"本书页面（http://www.ituring.com.cn/book/2007）下载书中示例代码。下载文件后，确保使用以下工具的最新版本来解压或提取文件夹：

- WinRAR / 7-Zip（Windows）
- Zipeg / iZip / UnRarX（Mac）
- 7-Zip / PeaZip（Linux）

勘误

虽然我们竭力确保图书内容的正确性，但错误在所难免。如果你在我们出版的任何一本图书中发现了文本或代码中的错误，希望你能告知我们，我们将非常感激。你的善举足以减少其他读者在阅读出错内容时的纠结和不快，并帮助我们在后续版本中更正错误。如果你发现任何错误，请通过“图灵社区”本书页面（http://www.ituring.com.cn/book/2007）告诉我们。一旦勘误通过确认，将显示在页面上的勘误表中。

侵权行为

所有媒体在互联网上都面临着侵权问题。我们严格保护自己的版权和许可证。如果你在互联网上发现有关我们出版物的任何形式的盗版产品，请立即告知我们地址或网站名称，以便我们进行补救。

请将盗版图书的网站地址发送到 ebook@turingbook.com。

你的反盗版行动就是在保护作者和出版社，只有这样，我们才能继续以优质内容回馈像你这样的热心读者。

问题

如果对本书存有任何方面的疑问，可以通过“图灵社区”本书页面（http://www.ituring.com.cn/book/2007）联系我们，我们将尽力为你答疑解惑。

电子书

扫描如下二维码，即可购买本书电子版。

目 录

第 1 章 React 基础

你好！

本书假设你已经知道 React 是什么并且了解它能为你解决什么问题。你可能使用 React 开发过中小型应用，但希望进一步提升自己的技能并得到所有未解决问题的答案。

你应该知道，React 由 Facebook 的开发人员及 JavaScript 社区的数百名贡献者所维护。

React 是最流行的 UI 开发库之一，因高性能而为人所熟知，这得益于它操作 DOM 的方式很巧妙。

React 包含了全新的 JSX 语法，该语法用于在 JavaScript 中编写标记，这需要你重新思考关注点分离原则[①]。React 还包含了许多很棒的特性，如服务端渲染，该特性让你可以开发通用应用。

为了学习本书，你需要了解如何使用终端程序在 Node.js 环境中安装并运行 `npm` 包。

本书中的所有代码示例都遵循 ES2015 标准，以方便你阅读与理解。

本章会讲解你需要掌握的一些基本概念，了解这些概念有助于你高效使用 React。对于初学者来说，理解这些概念意义重大。

- 命令式编程与声明式编程的区别。
- React 组件与组件实例，以及 React 如何使用元素来控制 UI 流程。
- React 如何改变 Web 应用的开发方式，它主张的另一种全新的关注点分离概念是什么，它选择不寻常的设计理念的原因是什么。
- 为什么人们会对 JavaScript 框架感到疲劳？在 React 生态系统中，怎样避免开发人员最常犯的错误？

① 关注点分离是软件设计原则之一，前端开发中一般指文档结构、样式表现以及脚本行为的分离。——译者注

1.1　声明式编程

只要阅读过 React 文档或者相关博文，那么你肯定遇到过**声明式**这一术语。

其实，React 如此强大的原因之一就在于它推行声明式编程范式。

因此，要想掌握 React，就需要理解声明式编程的含义，以及其与命令式编程之间的主要区别。

理解该问题的最简方式是：命令式编程描述代码如何工作，而声明式编程则表明想要实现什么目的。

与命令式世界极其相似的一个真实示例就是去酒吧喝啤酒并对服务员做出以下指示：

- 从架子上拿一个玻璃杯；
- 将杯子放到酒桶前；
- 按下酒桶开关，将杯子倒满；
- 把杯子递给我。

但在声明式世界中，你只需要说："请给我一杯啤酒。"

按声明式方式点一杯啤酒，需要假设服务员知道如何提供啤酒，这是声明式编程工作原理的一个重要方面。

我们来看一个 JavaScript 代码的示例。编写一个简单函数，给定包含大写字符串的数组时，该函数返回包含相同的小写字符串的数组。

```
toLowerCase(['FOO', 'BAR']) // ['foo', 'bar']
```

解决该问题的命令式函数的实现如下所示：

```
const toLowerCase = input => {
  const output = []
  for (let i = 0; i < input.length; i++) {
    output.push(input[i].toLowerCase())
  }
  return output
}
```

首先，创建一个空数组来保存结果。接着，函数循环遍历输入数组中的所有元素，并将小写值推进空数组中。最后，返回需要输出的数组。

声明式实现如下所示：

```
const toLowerCase = input => input.map(
  value => value.toLowerCase()
)
```

输入数组中的元素会传递到 map 函数，然后 map 函数会返回包含小写值的新数组。

这里需要注意几点比较重要的差别：前一个示例不够优雅，而且需要花更多精力才能理解；后者更加简洁、易读，这对注重可维护性的大型代码库来说非常重要。

另外值得一提的是，声明式编程中无须使用变量，也不用在执行过程中持续更新变量的值。事实上，声明式编程往往避免了创建和修改状态。

我们来看最后一个示例，了解一下 React 的声明式具体指什么。

我们要解决的问题是 Web 开发中常见的一个需求：展示带有标记的地图。

JavaScript 的实现（使用 Google Maps SDK）如下所示：

```
const map = new google.maps.Map(document.getElementById('map'), {
  zoom: 4,
  center: myLatLng,
})

const marker = new google.maps.Marker({
  position: myLatLng,
  title: 'Hello World!',
})
marker.setMap(map)
```

这显然是命令式的，因为代码逐条描述了创建地图、创建标记以及在地图上添加标记的指令。

改用 React 组件在页面上显示地图的方式如下所示：

```
<Gmaps zoom={4} center={myLatLng}>
  <Marker position={myLatLng} title="Hello world!" />
</Gmaps>
```

使用声明式编程方法时，开发人员只需要描述他们想要实现什么目的，无须列出实现效果的所有步骤。

声明式编程方式使得 React 很容易使用，因此最终的代码也很简单，这样产生的 bug 也更少，可维护性也更强。

1.2　React 元素

本书假设你已经熟悉组件及其实例，但要想高效地使用 React，你还需要了解另一种对象：**元素**。

无论是调用 `createClass` 方法、继承 `Component` 类还是声明一个无状态函数，其实都是在创建组件。React 管理着运行环境中的所有组件实例，在某个特定时刻，内存中可能存在同一

个组件的多个实例。

如前文所述，React 遵循声明式范式，因此无须告诉它如何与 DOM 交互。你只要声明希望在屏幕上看到的内容，React 就会完成剩下的工作。

或许你之前体会过，大部分其他 UI 库都是按相反方式工作的：它们让开发人员负责更新界面，这就需要手动管理 DOM 元素的创建与销毁。

React 使用了**元素**这种特殊类型的对象来控制 UI 流程。元素描述了屏幕上需要显示的内容。这些不可变对象比组件和组件实例要简单得多，而且只包含了展示界面所必需的信息。

以下示例展示了一个元素：

```
{
  type: Title,
  props: {
    color: 'red',
    children: 'Hello, Title!'
  }
}
```

元素最重要的属性是 `type`，另一个比较特殊的属性是 `children`，它是可选的，用于表示元素的直接后代。当然，元素还具有其他一些属性。

`type` 属性很重要，因为它告诉 React 如何处理元素本身。实际上，如果 `type` 属性是字符串，那么元素就表示 **DOM 节点**；如果 `type` 属性是函数，那么元素就是**组件**。

DOM 元素和组件可以互相嵌套，以表示整个渲染树：

```
{
  type: Title,
  props: {
    color: 'red',
    children: {
      type: 'h1',
      props: {
        children: 'Hello, H1!'
      }
    }
  }
}
```

当元素的 `type` 属性是函数时，React 会调用它，传入 `props` 来取回底层元素。React 会一直对返回结果递归地执行相同的操作，直到取回完整的 DOM 节点树，然后就可以将它渲染到屏幕。这个过程称作**一致性比较**，React DOM 和 React Native 都利用它在各自的平台上创建 UI。

1.3　忘掉所学的一切

初次使用React需要持开放的心态，因为它带来了设计Web应用和移动应用的一种全新方式。实际上，React 试图打破我们熟知的大部分最佳实践，并革新 UI 的构建方式。

过去的 20 年里，我们学到了十分重要的关注点分离原则，并习惯性地将其理解为从模板中分离逻辑。我们的目标始终是将 JavaScript 和 HTML 写入不同文件。

各种各样的模板方案被创建出来，以帮助开发人员实现这个目标。

问题在于，这种形式的分离大多数情况下只是一种假象。真相是，无论 JavaScript 和 HTML 写在什么地方，它们都是紧密耦合的。

我们来看一个模板的示例：

```
{{#items}}
  {{#first}}
    <li><strong>{{name}}</strong></li>
  {{/first}}
  {{#link}}
    <li><a href="{{url}}">{{name}}</a></li>
  {{/link}}
{{/items}}
```

以上代码段摘自 Mustache 官网，它是最流行的模板系统之一。

第一行代码让 Mustache 循环遍历项目集合。循环内部包含了一些条件逻辑，以检查`#first`和`#link`属性是否存在，并根据它们的值渲染不同的 HTML 片段。变量位于双花括号中。

如果应用只需要显示一些变量，那么模板库就是很好的解决方案，但需要操作复杂的数据结构时，情况就不一样了。

实际上，模板系统及其领域专用语言（domain-specific language，DSL）提供了一个功能子集，它们试图提供一门真正编程语言的功能，而又不用像语言那样完备。

如以上示例所示，模板高度依赖从逻辑层接收到的数据模型来显示信息。

另一方面，JavaScript 操作模板渲染出的 DOM 元素来更新 UI，即使它们是从独立文件中加载的。

样式也存在同样的问题：它们定义在不同的文件中，但模板引用了样式文件，而且 CSS 选择器也遵循了文档标记结构，因此，几乎不可能在不影响其他文件的前提下修改某个文件。这就是耦合的定义。

这就是为何传统的关注点分离原则更像是技术分离，这种做法当然说不上不好，但没有解决

任何真正问题。

React 尝试更进一步，将模板放到其所属位置，即与逻辑在一起。React 这么做的理由在于，它建议你通过编写名为**组件**的小型代码块来组织应用。

框架不应该指导你如何分离关注点，因为每个应用都有自己的方式，只有开发人员才能决定如何划分应用的界限。

组件式开发彻底改变了 Web 应用的开发方式，这也是关注点分离这一经典概念逐渐被更现代化的架构所代替的原因。

React 所主张的范式并不新奇，也不是由 React 的开发人员所发明的，但是 React 为这个概念的主流化做出了巨大贡献。最重要的是，不同专业水平的开发人员都能够轻松理解这个概念，它也因此流行起来。

React 组件的渲染方法如下所示：

```
render() {
  return (
    <button style={{ color: 'red' }} onClick={this.handleClick}>
      Click me!
    </button>
  )
}
```

一开始都会觉得这种语法有些奇怪，不过这只是因为我们还没有习惯而已。

一旦学会，我们就会意识到它的强大之处并理解其潜力。

同时使用 JavaScript 来编写逻辑和模板不但有助于更好地分离关注点，也赋予我们更强的能力和表达力，这正是构建复杂 UI 所必需的。

所以，即使混用 JavaScript 和 HTML 的做法乍听起来很奇怪，也请花 5 分钟试用 React。

学习新技术的最佳方式是在小项目中试用，并观察具体的应用情况。一般来说，如果新技术能带来长远利益，那么正确的做法就是忘掉学过的一切并改变思维模式。

此外，开发 React 的工程师一直在社区中推广另一个概念：将样式的逻辑也放到组件中。这个概念颇具争议，而且很难被接受。

React 的最终目标是将创建组件所用到的每项技术都封装起来，并根据它们的领域和功能进行关注点分离。

以下示例展示了 React 文档中的一个样式对象：

```
var divStyle = {
  color: 'white',
```

```
  backgroundImage: 'url(' + imgUrl + ')',
  WebkitTransition: 'all', // 注意此处大写的'W'
  msTransition: 'all' // 'ms'是唯一小写的厂商前缀
};

ReactDOM.render(
 <div style={divStyle}>Hello World!</div>,
 mountNode
);
```

开发者使用 JavaScript 编写样式的这套方案称作 CSS in JavaScript。第 7 章将对其进行详细介绍。

1.4 常见误解

一种常见的观点认为，React 是一个庞大的技术和工具集，要想使用它，就必须与包管理器、转译器、模块打包器以及无数的库打交道。

这种观念相当普遍，人们口口相传，以至于它有了明确的定义，还被赋予了 JavaScript 疲劳这一名称。

理解这种观念背后的缘由并不难。实际上，React 生态系统中的所有代码仓库和类库都源自炫酷的新技术、最新版的 JavaScript 以及最先进的技术和范式。

此外，GitHub 上还有大量的 React 构建模板，每个模板都包含了数十个能够解决各种问题的依赖项。

这很容易让人觉得使用 React 就一定要使用这些工具，但事实并非如此。

虽然人们普遍持有以上观点，但 React 其实是一个很小的库。像之前使用 jQuery 或 Backbone 那样，我们可以在任何页面（甚至 JSFiddle）中使用它：只要在关闭主体元素前引入脚本即可。

实际需要引入两个脚本，因为 React 拆分成了两个包：核心包 `react` 实现了 React 库的核心特性，`react-dom` 则包含了与浏览器相关的所有特性。这样做的理由是，核心包可以用于支持不同的目标平台，如浏览器中的 React DOM 以及移动设备上的 React Native。

在单个 HTML 页面中运行 React 应用不需要任何包管理器和复杂操作。只需下载发行包并托管到自己的服务器上（或者使用 https://unpkg.com），你就可以在短短几分钟内开始使用 React 及其特性。

要想使用 React，可以在 HTML 代码中添加以下 URL：

- https://cdn.bootcss.com/react/15.3.2/react.min.js
- https://cdn.bootcss.com/react/15.3.2/react-dom.min.js

如果只引入核心包，则无法使用 JSX 语法，因为它不是浏览器支持的标准语言。不过，重点在于先从最小特性集入手，需要时再加入更多功能。

对于简单的 UI，只需使用 `createElement` 方法。只有当开发更复杂的 UI 时，才需要引入转译器来启用 JSX 语法并将其转换成 JavaScript。

一旦应用变得更大，就需要使用路由库来处理不同的页面和视图。与上同理，引入路由库即可。

有些情况下，我们需要从 API 路径加载数据。如果应用继续扩增，就需要利用外部依赖项对复杂操作进行抽象，此时才应该引入包管理器。

然后就需要将应用拆分成独立的模块并按正确的形式组织文件。此时应该开始考虑使用模块打包器。

使用这种非常简单的方法，就不会再感到疲劳了。

如果代码模板带有数百个依赖和数十个 `npm` 包，不熟悉它们必然会感到不知所措。

值得注意的是，与编程相关的每种工作（特别是前端工程）都需要持续的学习。Web 的本质决定了它要根据用户与开发者的需求快速进化与改变。我们的生态环境从一开始就按这种方式发展，这也正是它如此令人激动的原因。

积累了一些 Web 开发经验后，我们了解到掌握一切知识是不可能的，但应该用正确的方法学习新知识，以避免疲劳感。只要能跟上所有的新趋势，就不需要为了掌握新类库而实际运用它，除非我们有时间做业余项目。

JavaScript 领域很令人吃惊：只要发布或者起草了一个规范，社区中就会有人以转译器插件或腻子脚本的形式实现它，这使得在浏览器厂商赞成并开始支持该规范前，大家就可以使用它。

以上事实使得 JavaScript 和浏览器生态环境与任何其他语言或平台完全不同。

其弊端是一切变化得太快，不过问题也只是如何在押宝新技术和保持稳妥间找到平衡。

任何情况下，Facebook 的开发人员都很注重开发体验，并且善于倾听社区反馈。因此，尽管使用 React 并非一定要学习数百种工具，但他们也意识到了这种令人疲劳的现象，于是发布了一个 CLI 工具，让构建和运行真正的 React 应用变得非常简单。

这个 CLI 工具只需要 Node.js/npm 环境，然后就可以全局安装：

```
npm install -g create-react-app
```

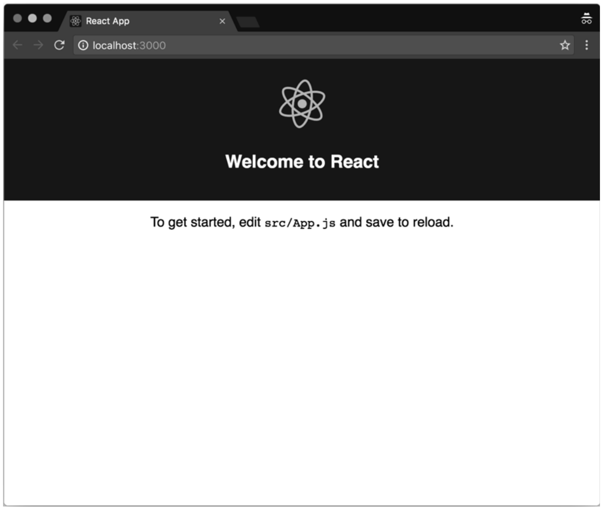
安装好这个可执行程序后，就可以向它传递文件夹名称来创建应用了。

```
create-react-app hello-world
```

最后，执行 `cd hello-world` 命令进入应用的文件夹，接着运行以下命令：

```
npm start
```

应用神奇地只靠一项依赖就可以运行了，但包括了用最先进的技术开发完整 React 应用所需的一切特性。以下截图展示了 `create-react-app` 创建的应用的默认页面。



本书将利用这个工具运行每章中的代码示例，你也可以在本书主页中获取它们：http://www.ituring.com.cn/book/2007。

1.5 小结

我们在本章中学习了一些基本概念，它们对于学习后续章节很重要，对于平时开发 React 应用也至关重要。

现在我们知道了如何编写声明式代码,并清晰地理解了自己开发的组件与 React 元素的区别，React 元素的实例会显示在屏幕上。

我们了解了将逻辑和模板放在一起的原因，以及为什么这种不太流行的决策能为 React 带来巨大成功。

我们深入探讨了为什么 JavaScript 生态系统中的人们会普遍感到疲劳，也研究了如何通过迭代的方式来避免这些问题。

最后，我们探讨了新的 CLI 工具 `create-react-app`。现在是时候动手编写一些真正的代码了。

第 2 章

整理代码

本章假设你已经具有 JSX 语法的使用经验，并且希望提升自己的技能以高效使用它。

要想使用 JSX 时不产生任何问题或不可预测的行为，重点要理解它的内部工作原理及其在 UI 构建上用处很大的原因。

我们的目的是编写整洁且可维护的 JSX 代码，为了实现该目的，我们需要了解其起源、怎样转换为 JavaScript 以及有哪些特性。

一开始我们会进行回顾，请耐心一点，因为掌握好基础对应用最佳实践至关重要。

本章内容具体如下所示。

- JSX 是什么，为什么要使用 JSX。
- Babel 是什么，怎样利用它来编写现代 JavaScript 代码。
- JSX 的主要特性以及其与 HTML 之间的区别。
- 编写优雅且可维护的 JSX 代码的最佳实践。
- 代码检查（尤其是 ESLint）怎样使得多个应用或团队的 JavaScript 代码风格保持一致。
- 函数式编程的基础，以及为何遵循函数范式可以使得我们写出更好的 React 组件。

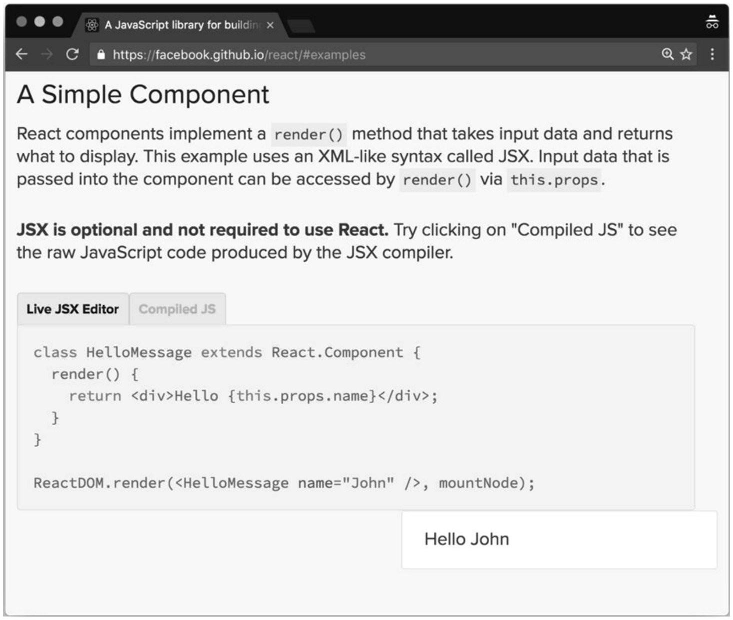
2.1 JSX

我们在第 1 章中看到 React 融合了组件内部的技术界限，改变了关注点分离的概念。

我们还学习了 React 如何使用组件返回的元素来展示屏幕上的 UI。

现在我们来看看如何在组件内声明元素。

React 提供了两种定义元素的方式。一种是使用 JavaScript 函数，另一种是使用类似 XML 的 JSX 语法。以下是 React.js 官网的示例：



一开始就要接触 JSX 是阻碍人们进一步了解 React 的主要原因之一，因为大多数人看到官网首页示例将 JavaScript 和 HTML 写在一起都会觉得很奇怪。

只要习惯了这种语法，就会认识到使用它很方便，因为它和 HTML 很相似，编写过 Web 平台 UI 的人都会觉得很熟悉。

开始和闭合标签可以使得嵌套元素树的表示变得非常简单，但如果用普通的 JavaScript 语法来做，那么嵌套元素树会变得难以阅读和维护。

2.1.1 Babel

为了在代码中使用 JSX（及 ES2015 的特性），我们需要安装 Babel。

首先，需要清楚地理解 Babel 能为我们解决什么问题，以及为什么需要在开发流程中增加这个步骤。原因是我们想在浏览器这个目标环境中使用尚未实现的语言特性。这些高级特性允许开发者编写更整洁的代码，但浏览器无法识别并执行。

解决方案就是用 JSX 和 ES2015 语法编写脚本，准备发布时再将代码编译成当今主流浏览器都已实现的 ES5 标准规范。

Babel 是 React 社区广泛使用的一个流行 JavaScript 编译器。

Babel 可以将 ES2015 的 JavaScript 代码编译成 ES5 的，也可以将 JSX 编译成 JavaScript 函数。这个过程称为转译，因为它将源代码编译成另一份新源代码，而不是可执行文件。

使用 Babel 相当简单，安装即可。

```
npm install --global babel-cli
```

如果不想全局（开发人员往往不喜欢）安装 Babel，也可以将它安装到项目中，然后通过 npm 脚本来运行，但考虑到本章的目的，需要暂且采用全局安装的做法。

安装完成后，可以运行以下命令来编译任何 JavaScript 文件。

```
babel source.js -o output.js
```

Babel 如此强大的原因之一在于可以灵活配置。它只是一个将源文件转译成输出文件的工具，配置后才能使用一些转换规则。

好在有很多非常有用的预设配置，其安装和用法都非常简单：

```
npm install --global babel-preset-es2015
babel-preset-react
```

安装完成后，在根文件夹下创建名为.babelrc 的配置文件，并写入以下代码来告诉 Babel 使用这些预设配置。

```
{
  "presets": [
    "es2015",
    "react"
  ]
}
```

从现在起，我们就可以用 ES2015 和 JSX 来编写代码源文件，并在浏览器中运行输出文件了。

2.1.2　Hello，World！

支持 JSX 的环境搭建好后，我们就可以深入学习最基础的示例：生成 div 元素。

以下代码展示了如何用 React 的 createElement 函数创建 div 元素：

```
React.createElement('div')
```

以下是 JSX 写法：

```
<div />
```

它看起来很像普通的 HTML。

最大的区别在于我们将标记写在了.js 文件中，值得注意的是，JSX 仅仅是语法糖，在浏览器

中运行前需要转译成 JavaScript。

实际上，运行 Babel 时会将<div />转换成 React.createElement('div')，编写模板时要始终牢记这一点。

2.1.3 DOM 元素与 React 组件

有了 JSX 后，我们既可以创建 HTML 元素，也可以创建 React 元素；唯一的区别在于它们是否以大写字母开头。

例如，渲染 HTML 按钮元素时使用<button />，而渲染 Button 组件时使用<Button />。

前一个按钮会转译为以下代码：

```
React.createELement('button')
```

后一个按钮会转译为以下代码：

```
React.createElement(Button)
```

以上区别在于，前一个调用传入了字符串形式的 DOM 元素类型，而后者传入了组件本身，这也意味着该组件要存在于当前作用域。

你可能已经注意到，JSX 支持自闭的标签，这样可以很好地保持代码简洁，无须重复编写不必要的标签。

2.1.4 属性

JSX 可以非常方便地书写包含属性的 DOM 元素或 React 组件。实际上，用 XML 设置元素属性就很简单。

```
<imgsrc="https://facebook.github.io/react/img/logo.svg"
alt="React.js" />
```

JavaScript 的等效写法如下所示：

```
React.createElement("img", {
  src: "https://facebook.github.io/react/img/logo.svg",
  alt: "React.js"
});
```

以上代码的可读性较差，虽然只有几个属性，但也需要经过一番思考才能读懂。

2.1.5 子元素

JSX 允许定义子元素来描述元素树，并构建复杂的 UI。

以下是一个简单的文本链接示例：

```
<a href="https://facebook.github.io/react/">Click me!</a>
```

它会转译为以下代码：

```
React.createElement(
  "a",
  { href: "https://facebook.github.io/react/" },
  "Click me!"
);
```

出于布局的需要，链接可以包裹在 div 元素的内部，而 JSX 代码段如下所示：

```
<div>
  <a href="https://facebook.github.io/react/">Click me!</a>
</div>
```

等效的 JavaScript 代码如下所示：

```
React.createElement(
  "div",
  null,
  React.createElement(
    "a",
    { href: "https://facebook.github.io/react/" },
    "Click me!"
  )
);
```

现在，类似 XML 的 JSX 代码拥有更好的可读性和可维护性，但了解 JSX 所对应的 JavaScript 代码非常重要，这样我们才能熟练掌握元素的创建。

JSX 的妙处在于没有限制只能将元素嵌套为其他元素的子元素，还可以使用函数或变量这样的 JavaScript 表达式。

要想这样做，只需要用双花括号括起表达式即可：

```
<div>
  Hello, {variable}.
  I'm a {function()}.
</div>
```

同理，这也适用于非字符串属性：

```
<a href={this.makeHref()}>Click me!</a>
```

2.1.6　JSX 与 HTML 的区别

到目前为止，我们只探讨了 JSX 和 HTML 之间的相似之处。现在我们来了解一下两者间的

微小区别，以及为什么会有这些区别。

1. 属性

我们要始终牢记，JSX 不是一门标准语言，需要转译成 JavaScript。由于这一点，有些属性无法使用。

比如，我们需要用 `className` 取代 `class`，用 `htmlFor` 取代 `for`：

```
<label className="awesome-label" htmlFor="name" />
```

这是因为 `class` 和 `for` 都是 JavaScript 的保留字。

2. 样式

非常明显的区别之一就是样式属性的工作原理。我们将在第 7 章中介绍更多细节，目前只需要了解其工作原理即可。

与 HTML 不同，样式属性期望传入 JavaScript 对象，而不是 CSS 字符串，而且样式名的写法为**驼峰式命名法**：

```
<div style={{ backgroundColor: 'red' }} />
```

3. 根元素

JSX 和 HTML 之间还有一个很重要的区别值得一提，因为 JSX 元素会转换为 JavaScript 函数，但 JavaScript 不允许返回两个函数，因此如果有多个同级元素，需要强制将它们封装在一个父元素中。

观察以下这个简单示例：

```
<div />
<div />
```

上述代码会导致以下错误：

```
Adjacent JSX elements must be wrapped in an enclosing tag
```

而以下写法就是有效的：

```
<div>
  <div />
  <div />
</div>
```

必须添加多余的 `div` 标签使得 JSX 生效这个操作无疑让人感到恼火，不过 React 的开发人员目前（撰写本书之时）正在寻找解决方案：

https://github.com/reactjs/core-notes/blob/master/2016-07/july-07.md

4. 空格

一开始让人觉得麻烦的还有一点，我们要始终记住 JSX 不是 HTML 这个事实，尽管它的语法很像 XML。

实际上，JSX 处理文本和元素间的空格的方式与 HTML 不同，这种方式有点违反直觉。

查看以下代码片段：

```
<div>
  <span>foo</span>
  bar
  <span>baz</span>
</div>
```

浏览器解析 HTML 时，以上代码会显示 `foo bar baz`，这与我们的预想相同。

而 JSX 会将同一份代码渲染为 `foobarbaz`，这是因为嵌套的三行代码转译成了 `div` 元素的独立子元素，没有将空格计算在内。为了得到与 HTML 一致的输出结果，普遍的解决方案是在元素间显式插入空格。

```
<div>
  <span>foo</span>
  {' '}
  bar
  {' '}
  <span>baz</span>
</div>
```

如你所见，这里用 JavaScript 表达式封装了空字符串来强制编译器在元素间插入空格。

5. 布尔值属性

开始真正学习在 JSX 中定义布尔值属性前，还需要了解一些基础知识。如果设置某个属性却没有赋值，那么 JSX 会默认其值是 `true`，这种行为类似 HTML 的 `disabled` 属性。

这意味着如果要将属性值设置为 `false`，则需要显式地声明。

```
<button disabled />
React.createElement("button", { disabled: true });
```

以下是另一个示例：

```
<button disabled={false} />
React.createElement("button", { disabled: false });
```

这一开始会让人感到疑惑，因为我们认为遗漏属性值应该为 `false`，而实际并非如此。在使用 React 时，我们应当始终显式地声明，以避免发生混淆。

2.1.7 展开属性

展开属性操作符也是一项很重要的特性，它来源于 ECMAScript 提案中的对象剩余/展开属性（Object Rest/Spread Properties for ECMAScript），该特性可以非常方便地为元素传递 JavaScript 对象的全部属性。

向子元素传递数据时，不要按引用方式传递整个 JavaScript 对象，而要使用对象的基本类型值以方便校验。这种做法很常见，并且引发的 bug 更少，写出的组件更稳健且更不易出错。

该特性的用法如下所示：

```
const foo = { id: 'bar' }
return <div {...foo} />
```

以上代码的转译结果如下所示：

```
var foo = { id: 'bar' };
return React.createElement('div', foo);
```

2.1.8 JavaScript 模板

最后，我们假设将模板移到组件内部而不用外部模板库具有一个优势，即可以利用 JavaScript 的完整功能，接下来我们探讨一下这个优势的具体意义。

展开属性就是一个示例，另一个明显的示例是可以用双花括号封装 JavaScript 表达式以作为属性值：

```
<button disabled={errors.length} />
```

2.1.9 常见模式

现在我们已经学习并掌握了 JSX 的原理，接下来将了解如何遵循一些有用的约定和技巧，以便正确使用 JSX。

1. 多行书写

我们先来看一个简单模式。前文提过，应该倾向于使用 JSX 而不是 `createElement` 方法，主要原因之一便是 JSX 的语法很像 XML，而且对称的开闭标签可以完美地表示节点树。

因此，我们应该尝试掌握它的正确用法并加以充分利用。

参见以下示例；需要嵌套元素的任何情况下都应该多行书写：

```
<div>
  <Header />
```

```
  <div>
    <Main content={...} />
  </div>
</div>
```

这比以下写法更易读：

```
<div><Header /><div><Main content={...} /></div></div>
```

如果出现子节点不是元素，而是文本或变量这样的例外情况，那么应该和父节点的标签写在同一行，并避免产生混淆，具体代码如下所示：

```
<div>
  <Alert>{message}</Alert>
  <Button>Close</Button>
</div>
```

多行书写元素时，一定要记得用括号封装它们。JSX 本质上会替换成函数，由于自动分号插入机制的存在，另起一行的函数可能会导致意外结果。例如，在渲染方法内返回 JSX 代码，这也是 React 创建 UI 的方式。

以下示例可以正常运行，因为 `div` 元素和返回在同一行：

```
return <div />
```

但接下来的示例就失效了：

```
return
  <div />
```

因为它会转换为以下代码：

```
return;
React.createElement("div", null);
```

因此你需要将代码语句包裹在括号内：

```
return (
  <div />
)
```

2. 多个属性的书写

编写 JSX 代码经常遇到的一个问题是元素拥有多个属性。一种方案是将所有属性写在同一行，但这样会使得一行代码变得特别长，我们不希望代码出现这种情况（后文介绍了如何强制执行代码风格指南）。

常见的解决方案是一行书写一个属性，同时缩进一个层级，并保持结尾括号和开始标签对齐：

```
<button
  foo="bar"
  veryLongPropertyName="baz"
  onSomething={this.handleSomething}
/>
```


3. 条件语句

当用到**条件语句**时，情况就变得很有趣了。例如，我们只想在满足特定条件时渲染一些组件。实际上，能够使用 JavaScript 判断条件已经具有很大优势了，不过 JSX 有许多不同方式来表达条件逻辑，理解每种方式的益处及其存在的问题对于编写可读且可维护的代码非常重要。

假设我们想要在用户当前登录到应用时显示注销按钮。

起初的代码如下所示：

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
}
return <div>{button}</div>
```

上述做法可行，但可读性不够好，组件和条件很多时会更差。

JSX 可以利用行内条件来判断：

```
<div>
  {isLoggedIn && <LoginButton />}
</div>
```

上述写法同样有效，因为如果条件为 `false`，则不会渲染任何组件，而如果条件为 `true`，那么 `LoginButton` 组件的 `createElement` 方法会被调用，并返回元素以构建最终的元素树。

如果条件语句有额外分支（常见的 `if...else` 语句），并且我们想要在用户登录后显示注销按钮，否则显示登录按钮，就可以利用 JavaScript 的 `if...else` 语句，如下所示：

```
let button
if (isLoggedIn) {
  button = <LogoutButton />
} else {
  button = <LoginButton />
}
return <div>{button}</div>
```

更好的替代方案是三元条件运算，因为代码更简洁：

```
<div>
  {isLoggedIn ? <LogoutButton /> : <LoginButton />}
</div>
```

三元条件运算在 Redux 等流行库所提供的真实示例（https://github.com/reactjs/redux/blob/master/examples/real-world/src/components/List.js#L25）中随处可见，组件获取数据时，示例根据 isFetching 变量的值使用三元运算符来显示按钮文本为 Loading 或者 load more。

```
<button [...]>
  {isFetching ? 'Loading...' : 'Load More'}
</button>
```

我们来看看更复杂情况下的最佳方案。例如，我们需要检查多个变量才能判断是否要渲染组件：

```
<div>
  {dataIsReady && (isAdmin || userHasPermissions) &&
    <SecretData />
  }
</div>
```

上述示例中的行内条件语句的写法很好，但可读性受到了很大影响。此时可以在组件内编写一个辅助函数来检验 JSX 的条件语句：

```
canShowSecretData() {
  const { dataIsReady, isAdmin, userHasPermissions } = this.props
  return dataIsReady && (isAdmin || userHasPermissions)
}

<div>
  {this.canShowSecretData() && <SecretData />}
</div>
```

如上所示，修改后的代码大大提升了可读性，条件语句也更直观。即使大半年后再回头看这段代码，也能够根据函数名清晰地看懂用途。

如果不喜欢用函数，那么你可以利用对象的 getter 方法使代码更优雅。

我们定义 getter 方法来取代函数，如下所示：

```
get canShowSecretData() {
  const { dataIsReady, isAdmin, userHasPermissions } = this.props
  return dataIsReady && (isAdmin || userHasPermissions)
}

<div>
  {this.canShowSecretData && <SecretData />}
</div>
```

同样的做法也可以用于计算属性。假设有两个独立属性 currency 和 value。除了将价格字符串写在渲染方法中，还可以创建一个类函数：

```
getPrice() {
  return `${this.props.currency}${this.props.value}`
}
```

```
<div>{this.getPrice()}</div>
```

这样做更好，因为单独抽离了生成字符串的代码，而测试包含逻辑的代码更方便。

再进一步，也可以像前面那样用 getter 取代函数：

```
get price() {
  return `${this.props.currency}${this.props.value}`
}

<div>{this.price}</div>
```

回到条件语句的讨论，还有一些方案需要用到外部依赖。为了尽量小化应用包体积，最好避免引入外部依赖，不过当前这种特殊情况值得这样做，因为改进模板的可读性有很大好处。

第一项方案是 render-if，可以执行以下命令来安装：

```
npm install --save render-if
```

然后就可以轻松地在项目中使用它，如下所示：

```
const { dataIsReady, isAdmin, userHasPermissions } = this.props
const canShowSecretData = renderIf(
  dataIsReady && (isAdmin || userHasPermissions)
)

<div>
  {canShowSecretData(<SecretData />)}
</div>
```

我们将条件语句封装进 renderIf 函数。

这个工具函数的返回值也是一个函数，可以接收 JSX 标记作为参数，当条件为 true 时显示。

始终牢记，不要在组件内添加过多逻辑。有些组件可能需要这样做，但我们应该尽可能保持组件简洁易懂，这样便于定位和修复问题。

至少应该保持 renderIf 函数简洁，为了实现这个目的，可以使用另一个工具库 react-only-if，有了它之后，可以通过高阶组件来设置条件函数，只需要按照条件为真的情况编写组件即可。

第 4 章将介绍高阶组件，不过目前你只需要知道它们就是函数，可以接收组件并对其进行改进再返回，改进包括添加属性或修改行为。

可以执行以下命令来安装这个库：

```
npm install --save react-only-if
```

安装完成后，就可以按照以下方式在应用中使用它。

```
const SecretDataOnlyIf = onlyIf(
  ({ dataIsReady, isAdmin, userHasPermissions }) => {
    return dataIsReady && (isAdmin || userHasPermissions)
  }
)(SecretData)

<div>
  <SecretDataOnlyIf
    dataIsReady={...}
    isAdmin={...}
    userHasPermissions={...}
  />
</div>
```

在以上代码中，组件内部不包含任何逻辑。

将条件语句作为 `onlyIf` 函数的第一个参数传入，满足条件时就渲染组件。

用于校验条件的函数可以接收组件的属性、状态以及上下文环境。

这样就可以避免条件语句对组件造成污染，有助于我们更轻松地理解并探究组件的代码。

4. 循环

开发 UI 时经常需要展示列表。将 JavaScript 作为模板语言来展示列表非常方便。

如果在 JSX 模板中编写一个函数并返回数组，那么数组的每一项都会编译为一个元素。

前文提过，可以在花括号内使用任何 JavaScript 表达式，针对给定对象的数组生成元素数组，最常用的做法是使用 `map` 方法。

我们来深入探讨一个真实示例。假设你有一张用户列表，其中每个用户都有一个对应的姓名属性。

用以下代码创建一张无序用户列表：

```
<ul>
  {users.map(user =><li>{user.name}</li>)}
</ul>
```

这段代码简单而又强大，因为它结合了 HTML 和 JavaScript 两者的能力。

5. 控制语句

UI 模板中的条件和循环都是常见操作，使用 JavaScript 的三元操作符或 `map` 方法来实现它们看上去有些奇怪。JSX 的开发理念就是如此，它只抽象了元素的创建部分，而逻辑部分则留给真正的 JavaScript，这种做法很巧妙，但有时代码会不够简洁。

总的来说，我们的目的是从组件中移除所有逻辑，尤其是渲染方法中的。但有时需要根据应用的状态来显示或隐藏元素，经常还需要遍历集合与数组。

如果你认为用 JSX 完成此类需求可以提高代码可读性，可以直接用现成的 Babel 插件：jsx-control-statements。

我们来看看如何使用它。

首先安装：

```
npm install --save jsx-control-statements
```

安装完成后，将它添加到.babelrc 文件中的 Babel 插件列表。

```
"plugins": ["jsx-control-statements"]
```

接着就可以使用这个插件提供的语法了，Babel 会将它连同普通的 JSX 语法一同转译。

以下是使用该插件编写的条件语句：

```
<If condition={this.canShowSecretData}>
  <SecretData />
</If>
```

它会转译为三元表达式，如下所示：

```
{canShowSecretData ? <SecretData /> : null}
```

If 组件非常有用，但如果渲染方法中需要嵌套条件，那么它很容易变得混乱且难以理解。查看以下 Choose 组件的代码：

```
<Choose>
  <When condition={...}>
    <span>if</span>
  </When>
  <When condition={...}>
    <span>else if</span>
  </When>
  <Otherwise>
    <span>else</span>
  </Otherwise>
</Choose>
```

注意！上述代码会转译为多个三元表达式。

最后，我们介绍一个可以轻松实现循环的**组件**（记住，我们所提到的并非真实的组件，而只是语法糖）：

```
<ul>
  <For each="user" of={this.props.users}>
```

```
    <li>{user.name}</li>
  </For>
</ul>
```

上述代码会转译为 map 方法，这并没有什么神奇之处。

如果习惯使用 linter（一种代码检查工具），你可能会疑惑为何 linter 没有针对这些代码报错。实际上，user 变量在转译前并不存在，也没有封装在某个函数中。为了避免代码检查时报错，我们需要安装另一个插件：eslintplugin-jsx-control-statements。

如果无法理解上一段内容，不要担心；我们将在 2.2 节介绍代码检查。

6. 次级渲染

值得强调的是，我们总是希望组件可以足够小，渲染方法也要简单明了。

然而，实现这个目的并不简单，尤其是迭代开发应用时，我们无法在第一次迭代过程中准确地判断如何将组件拆分得更小。

那么当渲染方法的代码量多到难以维护时，应该做什么呢？一种方案是将其拆分成更小的方法，同时又将所有逻辑都保留在原有组件内部。

查看以下示例：

```
renderUserMenu() {
  // JSX 用于用户菜单
}

renderAdminMenu() {
  // JSX 用于管理员菜单
}

render() {
  return (
    <div>
      <h1>Welcome back!</h1>
      {this.userExists && this.renderUserMenu()}
      {this.userIsAdmin && this.renderAdminMenu()}
    </div>
  )
}
```

这种方案并不总是可以当作最佳实践，因为显然拆分组件的做法更好。有时这样做只是为了保持渲染方法简洁。Redux 的一个真实示例就是用一个次级渲染方法来渲染加载更多按钮。

现在我们已经熟练掌握了 JSX，接下来将继续深入学习如何在代码中遵循一套风格指南，以确保代码风格保持一致。

2.2 ESLint

我们总是希望尽可能写出最佳代码，但有时总会出错，然后需要花数小时定位 bug，最后发现只是拼写错误，这很令人沮丧。好在一些工具可以帮助我们在输入过程中检查代码的正确性。

这些工具无法表明代码能否实现预期效果，但可以帮助我们避免语法错误。

如果之前使用过 C#这种静态语言，那么你应该很熟悉 IDE 给出的这种警告信息。

Douglas Crockford 开发的 JSLint（最初发布于 2002 年）使得 JavaScript 代码检查变得流行起来。后来出现了 JSHint，如今 ESLint 成为了 React 领域的事实标准。

ESLint 是 2013 年发布的开源项目，由于其配置化程度高且扩展性良好，逐渐流行起来。

在 JavaScript 生态系统中，各种库和技术都变化迅速，因此关键是要找到一个可以方便地使用插件来扩展的工具，并且可以按需启用或禁用规则。

最重要的是，如今我们普遍使用 Babel 这样的转译器，以及尚未归入 JavaScript 标准版本的试验特性，因此需要让 linter 知道源代码文件遵循了哪些规则。

linter 不仅能帮助我们更少犯错，或者至少更早发现错误，它还能强制推行一些常见的编程风格指南。这一点非常重要，尤其是开发者众多的大型团队中的每个人都有自己偏爱的编程风格。

如果以不同风格编写代码库中的不同文件，甚至不同函数，那么这些代码将难以阅读。

2.2.1 安装

首先，执行以下命令来安装 ESLint：

```
npm install --global eslint
```

可执行程序安装完成后，就可以用以下命令来运行它：

```
eslint source.js
```

输出结果会告诉我们文件中是否有错。

安装后首次运行不会看到任何报错，因为它各方面都需要配置，一开始并不包含任何默认规则。

2.2.2 配置

现在我们开始配置 ESLint。

可以使用位于项目根目录的.eslintrc 文件来配置 ESLint。

使用 `rules` 属性来添加规则。

举例来说，创建.eslintrc 文件并禁用分号：

```
{
  "rules": {
    "semi": [2, "never"]
  }
}
```

上述配置文件的含义是："semi"是规则名，[2, "never"]是规则的值。一开始看到这种配置会觉得不够直观。

ESLint 规则具有决定问题严重程度的三个等级。

- off（或者 0）：禁用规则
- warn（或者 1）：规则会产生警告
- error（或者 2）：规则会抛出错误

将规则的值设为 2，因为我们希望当代码不符合规则时，ESLint 会抛出错误。

第二个参数将 ESLint 配置为不允许代码中使用分号（相反值为 `always`）。

ESLint 及其插件都有详细的文档，你可以找到任意一条规则的描述及其通过或失败的示例。

现在新建一个文件并写入以下代码。

```
var foo = 'bar';
```

（注意，此处使用了 `var` 关键词，因为 ESLint 还不知道我们要用 ES2015 语法来编码。）

执行 `eslint index.js` 后，就会看到以下提示：

```
Extra semicolon (semi)
```

这太棒了！linter 搭建完毕，它帮助我们遵循了第一条规则。

可以手动启用或禁用每条规则，也可以一步启用推荐配置，只需要在.eslintrc 中添加以下代码即可。

```
{
  "extends": "eslint:recommended"
}
```

`extends` 属性表明我们将沿用 ESLint 的推荐配置，另外我们也可以手动修改.eslintrc 的 `rules` 属性来覆盖每条规则，正如前文所做的那样。

启用推荐规则后，再次运行 ESLint，此时不会看到与分号相关的任何报错（推荐配置中不包

括这个部分），但 linter 会提示声明过的 `foo` 变量从未使用。

`no-unused-vars` 规则对于保持代码简洁非常有用。

一开始提过，我们希望用 ES2015 语法编写代码，但是以下代码会报错：

```
const foo = 'bar'
```

报错信息如下所示：

```
Parsing error: The keyword 'const' is reserved
```

因此，要想启用 ES2015 语法，需要添加配置选项：

```
"parserOptions": {
  "ecmaVersion": 6,
}
```

添加完毕后，就只剩下变量未使用的报错提示了，这是正常的。

最后使用以下配置来启用 JSX 语法：

```
"parserOptions": {
  "ecmaVersion": 6,
  "ecmaFeatures": {
    "jsx": true
  }
},
```

如果你之前开发过 React 应用却从未使用 linter，现在要想学习规则并开始习惯，那么最好运行 ESLint 来检查源代码并修复所有问题。

用 ESLint 帮助我们编写更好的代码的方式有很多种。一种是前文的做法：在命令行中运行 ESLint，并得到一系列错误提示。

这种做法可行，但一直手动执行不够方便。更好的做法是在编辑器中加入检查流程，这样输入代码时就能立刻得到反馈。Sublime Text、Atom 以及其他流行的编辑器都提供了 ESLint 插件来实现这个目的。

在真实的开发场景中，手动运行 ESLint 或者让编辑器实时提供反馈非常有用，但是还不够，因为我们会遗漏某些错误或警告，甚至是直接无视。

为了避免代码库中出现未检查的代码，我们可以将 ESLint 作为开发流程中的一环。举例来说，可以在测试时执行检查，如果代码不符合检查规则，那么整个测试步骤就算失败。

另一个方案是在发起 pull request 前进行代码检查，这样在同事开始审查前还有机会整理代码。

2.2.3 React 插件

前文提过，ESLint 流行起来的主要原因是其可以用插件进行扩展，对我们最重要一个插件是 `eslint-plugin-react`。

ESLint 不需要任何插件就能解析 JSX（启用配置开关即可），但我们想要更多功能。例如，我们可能会想要推行前面章节中的某项最佳实践，并使得模板在多个开发人员及团队间保持一致。

要想使用该插件，需要先进行安装：

```
npm install --global eslint-plugin-react
```

安装完成后，在配置文件中添加以下代码，以便 ESLint 可以使用它：

```
"plugins": [
  "react"
]
```

如你所见，配置非常直观，没有任何复杂的地方。与 ESLint 一样，没有配置规则的情况下它什么都不会做，我们可以启用推荐配置来激活基础规则集。

在.eslintrc 文件中更新`"extends"`属性，如下所示：

```
"extends": [
  "eslint:recommended",
  "plugin:react/recommended"
],
```

如果出现编写错误，比如 React 组件的同一个属性写了两次，那么就会出现错误提示：

```
<Foo bar bar />
```

以上代码会返回如下所示的错误提示：

```
No duplicate props allowed (react/jsx-no-duplicate-props)
```

大量规则可以用于项目。我们来了解其中一部分，看看它们是如何帮助我们遵循最佳实践的。

正如第 1 章中所说，按照元素的树结构缩进 JSX 代码有助于提升可读性。

如果代码库及组件的缩进风格不一致，则会出现问题。

我们来查看一个示例，了解一下 ESLint 如何帮助团队的每个成员遵循风格指南，而又无须死记硬背。

注意，这种情况下的不正确缩进实际上不算错误，代码还是能够正常运行的；这只是一致性问题。

首先，激活以下规则：

```
"rules": {
  "react/jsx-indent": [2, 2]
}
```

第一个 2 表示如果代码不符合规则，则 ESLint 应该给出错误提示，第二个 2 则表示每个 JSX 元素应该缩进两个空格。因为 ESLint 不会做任何决定，所以启用哪条规则完全取决于你自己。甚至可以通过设置第二参数为 0 来选择无缩进风格。

编写以下代码：

```
<div>
<div />
</div>
```

ESLint 会给出以下报错信息：

```
Expected indentation of 2 space characters but found 0
(react/jsx-indent)
```

换行书写属性值时也有类似的规则来约束缩进。

前文介绍过，属性过多或过长时，较好的做法是换行书写。

要想推行属性根据元素名缩进两个空格的格式，启用以下规则即可：

```
"react/jsx-indent-props": [2, 2]
```

至此，如果属性没有缩进两个空格，那么 ESLint 就会报错。

问题在于，如何界定一行代码过长？多少个属性算多？每个开发人员都有不同的看法。ESLint 的 `jsx-max-props-per-line` 规则有助于维护一致性，这样每个组件的编写方式就相同了。

ESLint 的 React 插件提供的规则不仅有助于写出更优雅的 JSX 代码，也有助于写出更好的 React 组件。

举例来说，可以强制要求属性类型按照字母表顺序排列、使用未声明的属性时给出错误提示，或者要求尽量编写无状态的函数组件，而不要使用类（第 3 章将介绍两者的详细区别）等。

2.2.4 Airbnb 的配置

我们已经了解了 ESLint 如何通过静态分析来发现错误，以及如何促使我们在整个代码库中遵循一致的风格指南。

我们也见识到了 ESLint 的灵活之处，以及如何通过配置与插件来扩展它。

我们还学到了可以用推荐配置来激活一套基本规则集，无须手动完成这个繁琐的过程。

接下来我们再进一步了解它。

ESLint 的 `extends` 属性非常强大，因此你可以从第三方配置入手，再添加自己特有的规则。

React 领域最流行的配置之一莫过于 Airbnb 的那一套。Airbnb 的开发者按照 React 的最佳实践创建了一套规则集，你可以直接在代码库中使用，无须自己手动判断启用哪条规则。

要想使用这套配置，必须先安装一些依赖：

```
npm install --global eslint-config-airbnbeslint@^2.9.0 eslint-plugin-
jsx-a11y@^1.2.0 eslint-plugin-import@^1.7.0 eslint-plugin-react@^5.0.1
```

然后在.eslintrc 中添加以下配置：

```
{
  "extends": "airbnb"
}
```

接着就可以尝试执行 ESLint 来检查你的 React 源代码文件，可以看到代码是否符合 Airbnb 规则，以及这些规则是否适合你。

以上就是开始使用代码检查工具最简单也最常用的方式。

2.3　函数式编程基础

除了编写 JSX 时遵循最佳实践，并使用 linter 来加强代码一致性以更早发现错误，保持代码简洁的另一个方法是：遵循**函数式编程**风格。

正如第 1 章中所说，React 的声明式编程提升了代码的可读性。

函数式编程就是一种声明式范式，能够避免代码副作用，同时它推崇数据不可变，以便更易维护与考量代码。

接下来的部分并非要全面介绍函数式编程；而只是介绍 React 普遍使用的一些概念，希望你可以理解它们。

2.3.1　一等对象

JavaScript 的函数是**一等对象**，这意味着它们可以赋给变量，也可以作为参数传递给其他函数。

这时要介绍一下**高阶函数**的概念了。高阶函数接受一个函数作为参数，也可以传入其他参数，最后返回另一个函数。返回的函数通常会添加一些增强的特殊行为。

我们来查看一个简单示例，一个两数相加的函数在增强后先打印所有参数，再接着执行原先的逻辑：

```
const add = (x, y) => x + y
const log = func => (...args) => {
  console.log(...args)
  return func(...args)
}

const logAdd = log(add)
```

理解这个概念非常重要，因为 React 领域的一个常用模式是使用**高阶组件**，将组件当作函数，并为它们增加一些常用行为。第 4 章将介绍高阶组件以及其他模式。

2.3.2 纯粹性

编写纯粹函数是函数式编程的一个重要方面。React 生态系统经常会遇到这个概念，尤其是了解 Redux 这类库后。

函数的纯粹性到底指什么呢？

纯粹函数是指它不产生副作用，也就是说它不会改变自身作用域以外的任何东西。

举例来说，如果函数改变了应用状态、修改了上层作用域定义的变量，或者与 DOM 这样的外部实体发生了交互，那么该函数就是非纯粹函数。

非纯粹函数很难调试，而且大多数时候不可能多次调用它们并期望得到同样的结果。

以下展示的就是纯粹函数：

```
const add = (x, y) => x + y
```

它可以运行多次，并且总能得到同样的结果，因为没有将数据存储在其他地方，也没有修改任何东西。

以下展示的就是非纯粹函数：

```
let x = 0
const add = y => (x = x + y)
```

执行 `add(1)` 两次，但得到了两个不同的结果。第一次是 1，而第二次是 2，尽管我们是用同样的参数调用同一个函数。出现这种情况的原因在于每次执行都修改了全局状态。

2.3.3 不可变性

我们已经知道了如何编写不改变状态的纯粹函数，但需要修改变量值时应该怎么做呢？在函

数式编程中，函数不会修改变量值，而是创建新的变量，赋新值后再返回变量。操作数据的这种方式称为**不可变性**。

不能修改不可变值。

我们来查看以下示例：

```
const add3 = arr => arr.push(3)
const myArr = [1, 2]
add3(myArr) // [1, 2, 3]
add3(myArr) // [1, 2, 3, 3]
```

上述代码中的函数没有遵循不可变性，因为它修改了给定数组的值。另外，调用这个函数两次会得到不同结果。

可以用 concat 方法改写以上函数，使其满足不可变性。concat 方法会返回新数组，而且不会修改原数组：

```
const add3 = arr => arr.concat(3)
const myArr = [1, 2]
const result1 = add3(myArr) // [1, 2, 3]
const result2 = add3(myArr) // [1, 2, 3]
```

此时即便运行该函数两次，myArr 仍然保有初始值。

2.3.4　柯里化

柯里化是函数式编程的常用技巧。柯里化过程就是将多参数函数转换成单参数函数，这些单参数函数的返回值也是函数。我们通过一个示例来弄清这个概念。

我们从前文的 add 函数入手，将它转换成柯里化函数。

原先的写法如下所示：

```
const add = (x, y) => x + y
```

将其定义为以下写法：

```
const add = x => y => x + y
```

然后按以下方式使用它：

```
const add1 = add(1)
add1(2) // 3
add1(3) // 4
```

这种函数写法相当方便，因为传入第一个参数后，第一个值被保留起来，返回的第二个函数可以多次复用。

2.3.5　组合

最后，可以用于 React 的函数式编程中的另一个重要概念是组合。函数（和组件）可以结合产生新函数，从而提供更高级的功能与属性。

思考以下函数：

```
const add = (x, y) => x + y
const square = x => x * x
```

这两个函数可以组合创建一个新函数，用于两数相加，再对结果求平方：

```
const addAndSquare = (x, y) => square(add(x, y))
```

遵循这个范式就可以编写小而简单、易于测试的纯粹函数，然后再将它们组合起来使用。

2.3.6　函数式编程与 UI

最后需要学习的就是如何用函数式编程构建 UI，这也正是使用 React 的目的。

可以将 UI 看作传入应用状态的函数，如下所示：

```
UI = f(state)
```

我们希望这是一个幂等函数，即传入相同的应用状态时会返回同样的 UI。

使用 React 时，可以将创建 UI 的组件看作函数，后文中会讲解这一点。

组件可以组合形成最后的 UI，这也正是函数式编程的特性之一。

React 构建 UI 的方式和函数式编程原则有很多相似之处，了解得越多，也就能写出越好的代码。

2.4　小结

本章介绍了大量有关 JSX 工作原理的内容，以及如何在组件中正确使用 JSX。我们从基础语法入手，奠定扎实基础以掌握 JSX 及其特性。

第二部分将介绍 ESLint 及其插件如何帮助我们更快发现错误，以及怎样在代码库中强制推行一致的风格指南。

最后，我们介绍了函数式编程的基础，以理解开发 React 应用所需要的重要概念。

现在代码已经足够整洁，接下来我们将深入研究 React 并学习如何编写真正可复用的组件。

第 3 章

开发真正可复用的组件

要想开发真正可复用的组件，我们需要理解 React 提供的定义组件的多种方式，并知道如何根据具体情况进行选择。React 引入了一种新型组件，允许将组件定义为**无状态函数**。最为关键的是要理解这种组件，并了解何时以及为何需要使用它。

你可能已经使用过组件的内部状态，但仍然不太了解其使用时机以及可能产生的问题。最好的学习方式就是阅读代码示例，我们将从一个只有单一功能的组件入手，然后将其改造为可复用组件。

我们将先回顾基本概念，然后继续深入学习。本章末尾将创建一套可用的组件风格指南。

本章包含如下内容。

- 创建 React 组件的不同方式以及如何进行选择。
- 无状态函数式组件是什么，以及函数式组件与状态组件的区别。
- 状态的工作原理，以及何时应该避免使用它。
- 为什么为每个组件定义清晰的 prop 类型很重要，以及如何根据 prop 类型用 React Docgen 动态地生成文档。
- 将耦合组件改造为可复用组件的实例。
- 如何使用 React Storybook 创建可用的风格指南，以便为可复用组件提供文档。

3.1 创建类

第 1 章中介绍了 React 如何利用元素将组件显示到屏幕上。

现在我们来了解一下 React 定义组件的不同方式以及使用各方式的原因。

再次强调，本书假设你已经在中小型应用中使用过 React，也就是说，你应该具备开发组件的经验。

你可能已经根据 React 官网提供的示例，或者参照搭建项目的脚手架模板的风格选择过一种方式。

你应该清楚地了解 prop、状态和生命周期方法这些概念，因为本章不会详细介绍这些内容。

3.1.1　createClass 工厂方法

可以查看（编写本书时的）React 文档的第一个示例，它展示了如何用 React.createClass 来定义组件。

先从一段很简单的代码开始：

```
const Button = React.createClass({
  render() {
    return <button />
  },
})
```

以上代码创建了一个按钮组件，并且应用的其他组件也可以引用它。

可以用纯 JavaScript 对其进行改写，如下所示：

```
const Button = React.createClass({
  render() {
    return React.createElement('button')
  },
})
```

无须用 Babel 进行转译即可在任何地方运行以上代码，这样做有利于快速上手 React，不用花时间学习 React 生态系统中的各种工具。

3.1.2　继承 React.Component

定义 React 组件的第二种方式是使用 ES2015 语法的类。现代浏览器广泛支持 class 关键词，不过为了稳妥起见，可以用 Babel 对其进行转译。一般来讲，如果用 JSX 进行编程，那么 Babel 已经包含在工具栈中。

我们来看看如何用类重写上例中的按钮：

```
class Button extends React.Component {
  render() {
    return <button />
  }
}
```

这种全新的组件定义方式随着 React 0.13 发布，Facebook 的开发人员希望可以在社区中推广

这种方式。举例来说，Facebook 的员工 Dan Abramov（也是活跃的社区成员）在比较 createClass 与继承 Component 时说道：

> “标准化的 ES6 类是更好用的利器。”

Facebook 希望开发人员使用后者，因为这是 ES2015 标准的一项特性，但 createClass 工厂方法不是。

3.1.3　主要区别

除了语法方面的差异，上述两种方法还有以下几项主要区别。

我们将详细介绍这些区别，以便你可以掌握完整的信息，从而根据团队情况以及项目需求做出最佳选择。

1. prop

第一个区别在于如何定义组件期望接收的 prop 及其默认值。

本章后面会详细介绍 prop 的工作原理，目前我们先专注于如何定义它们。

createClass 方法需要在作为参数传入函数的对象内定义 prop，同时在 getDefaultProps 内返回默认值：

```
const Button = React.createClass({
  propTypes: {
    text: React.PropTypes.string,
  },

  getDefaultProps() {
    return {
      text: 'Click me!',
    }
  },

  render() {
    return <button>{this.props.text}</button>
  },
})
```

在以上代码中，我们用 propTypes 属性列出了能够传递给组件的所有 prop。

接着我们用 getDefaultProps 函数定义了 prop 的默认值，如果父组件中传递了 prop，那么对应的默认值会被覆盖。

如果想要用类实现同样的目的，则需要用到稍微不同的结构：

```
class Button extends React.Component {
  render() {
    return <button>{this.props.text}</button>
  }
}

Button.propTypes = {
  text: React.PropTypes.string,
}

Button.defaultProps = {
  text: 'Click me!',
}
```


因为**类属性**仍处于草案阶段（尚未成为 ECMAScript 标准的一部分），所以若想定义类的属性，则需要在创建类之后再写入属性。

由示例可见，`propTypes` 对象与使用 `createClass` 方式创建的一模一样。

在设置默认的 prop 时，我们原先用函数来返回默认属性对象，但在使用类时，需要在类上定义 `defaultProps` 属性，再将默认值对象赋给它。

使用类的好处在于，只需要在 JavaScript 对象上定义属性，无须使用 `getDefaultProps` 这种 React 特有的函数。

2. 状态

`createClass` 工厂方法和 `extends React.Component` 方法的另一个重大区别是，组件初始状态的定义方式不同。

和前面一样，使用 `createClass` 需要调用函数，而使用 ES2015 的类则需要设置实例的属性。

我们来看一个示例：

```
const Button = React.createClass({
  getInitialState() {
    return {
      text: 'Click me!',
    }
  },

  render() {
    return <button>{this.state.text}</button>
  },
})
```

`getInitialState` 方法期望返回一个对象，该对象包含每个状态属性的默认值。

如果用类来定义初始状态，则需要在类的构造器方法内设置实例的状态属性：

```
class Button extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      text: 'Click me!',
    }
  }

  render() {
    return <button>{this.state.text}</button>
  }
}
```

定义状态的这两种方式等效，不过使用类的好处与前面所说的一样，即无须使用 React 特有的 API，直接在实例上定义属性即可。

在 ES2015 中，若想在子类中使用 `this`，必须先调用 `super` 方法。React 还会将 props 对象传给父组件。

3. 自动绑定

`createClass` 有一项非常方便的特性，但该特性也会隐藏 JavaScript 的工作原理，从而造成误解，对于新手而言尤其如此。这项特性允许我们创建事件处理器，并且当调用事件处理器时，`this` 会指向组件本身。

第 6 章将介绍事件处理器的工作方式。目前，我们只专注于它们与当前组件的绑定方式。

查看以下的简单示例：

```
const Button = React.createClass({
  handleClick() {
    console.log(this)
  },

  render() {
    return <button onClick={this.handleClick} />
  },
})
```

`createClass` 允许我们按照以上方式设置事件处理器，这样一来，函数内部的 `this` 就会指向组件本身。这允许我们调用同一组件实例的其他方法。例如，调用 `this.setState()`或者其他方法所产生的结果都能符合预期。

现在我们来看看 `this` 在类中的差别以及如何才能实现同样的行为。按照以下方式继承 `React.Component` 并定义组件：

```
class Button extends React.Component {
  handleClick() {
```

```
    console.log(this)
  }

  render() {
    return <button onClick={this.handleClick} />
  }
}
```

点击按钮后，控制台输出的结果为 `null`。这是因为函数传给事件处理器后丢失了对组件的引用。

这并不意味着不能在类中使用事件处理器，只是我们需要手动绑定函数。

我们来看看哪些方案可供采纳，以及它们分别适合哪种场景。

你可能已经知道，ES2015 提供的箭头函数可以自动将当前的 `this` 绑定到函数体。

查看这段代码示例：

```
() => this.setState()
```

Babel 会将以上代码转译为以下代码：

```
var _this = this;

(function () {
  return _this.setState();
});
```

可以得知，解决**自动绑定**问题的一种可能方案就是使用箭头函数，如下所示：

```
class Button extends React.Component {
  handleClick() {
    console.log(this)
  }

  render() {
    return <button onClick={() => this.handleClick()} />
  }
}
```

这样做符合预期，也不会带来什么特殊问题。唯一的缺点在于，如果在意性能，那么就需要理解代码的本质。

实际上，在渲染方法中绑定函数会带来无法预料的副作用，因为每次渲染组件（应用在生命周期内会多次渲染组件）时都会触发箭头函数。

虽然在渲染方法内多次触发某个函数不太理想，但本身并没有什么问题。

问题在于，如果这个函数传递给子组件，那么子组件在每次更新过程中都会接收新的 prop。

这可能会导致低效的渲染，进而引发问题，对于纯粹组件而言尤其如此（第 9 章将讨论性能方面的问题）。

解决函数绑定问题的最佳方案是在构造器内进行绑定操作，这样即使多次渲染组件，它也不会发生任何改变。

```
class Button extends React.Component {
  constructor(props) {
    super(props)

    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    console.log(this)
  }

  render() {
    return <button onClick={this.handleClick} />
  }
}
```

就是这样，问题解决了！

3.1.4 无状态函数式组件

还有另一种定义组件的方式，它与前两种差别很大。

React 0.14 引入了这个方法。它十分强大，可以使得代码更易维护和复用。

我们先来了解这个方法的原理及功能，然后再探讨其适用场景。

它的语法相当简洁优雅，查看以下示例：

```
() => <button />
```

以上代码创建了一个空按钮，简洁的箭头函数语法使得其代码变得直观且极具表现力。如你所见，不需要使用 `createClass` 工厂方法或者继承 `Component`，定义返回结果为待显示元素的函数即可。

当然，可以在函数体内使用 JSX 语法。

1. props 与上下文

不能从父组件接收 props 对象的组件没有多大用处，而无状态函数式组件可以接收 props 对象作为参数：

```
props => <button>{props.text}</button>
```

此外，还可以使用更简洁的 ES2015 解构语法：

```
({ text }) => <button>{text}</button>
```

定义 props 后，像继承组件那样，无状态函数就可以通过 `propTypes` 属性来接收 props：

```
const Button = ({ text }) => <button>{text}</button>

Button.propTypes = {
  text: React.PropTypes.string,
}
```

无状态函数式组件也接收表示上下文的第二个参数。

```
(props, context) => (
  <button>{context.currency}{props.value}</button>
)
```

2. 关键词 `this`

无状态函数式组件与状态组件的一项区别在于，`this` 在无状态函数式组件的执行过程中不指向组件本身。

由于这个原因，与组件实例相关的 `setState` 等方法以及生命周期方法都无法使用。

3. 状态

顾名思义，无状态函数式组件没有任何内部状态，这正是因为 `this` 不存在所导致的。这使得无状态函数式组件无比强大，同时又很容易使用。

无状态函数式组件只接收 props（以及上下文）参数，并返回相应元素。这体现了第 2 章中提到的**函数式编程**的原则。

4. 生命周期

无状态函数式组件没有提供任何像 `componentDidMount` 这样的生命周期钩子；它们只实现了一个类似渲染的方法，并将其他工作都交由父组件来执行。

5. ref 与事件处理器

因为无状态函数式组件不能访问组件实例，所以如果要使用 ref 或者事件处理器，需要按以下方式来定义。

```
() => {
  let input

  const onClick = () => input.focus()

  return (
    <div>
```

```
      <input ref={el => (input = el)} />
      <button onClick={onClick}>Focus</button>
    </div>
  )
}
```

6. 没有组件引用

无状态函数式组件的另一个不同点在于，无论何时使用 ReactTestUtils（第 10 章将详细介绍测试）来渲染它们，都无法取回对组件的引用。

例如：

```
const Button = React.createClass({
  render() {
    return <button />
  },
})

const component = ReactTestUtils.renderIntoDocument(<Button />)
```

在以上示例中，组件表示 Button。

```
const Button = () => <button />
const component = ReactTestUtils.renderIntoDocument(<Button />)
```

但这个示例中的组件为 null，将组件包裹在一个<div>标签中是一种解决方法，如下所示。

```
const component = ReactTestUtils.renderIntoDocument
(<div><Button/></div>)
```

7. 优化

使用无状态函数式组件需要牢记一点：虽然 Facebook 的开发人员宣称以后会为无状态组件提供性能优化，但在编写本书时，他们还没有明显的行动。

实际上，因为没有 shouldComponentUpdate 方法，所以无法通知 React 只在 props（或某个特定 prop）变化时才渲染函数式组件。

虽然这不是什么大问题，但也值得考虑。

3.2　状态

我们已经学习了如何用工厂方法、继承 React 类或者无状态函数式组件来创建组件。

现在我们来深入学习与状态相关的主题，了解为何使用它极其重要并弄清其工作原理。

我们将学习何时应该使用无状态函数，而不是状态组件，以及为何这代表了组件设计的一项基本决策。

3.2.1　外部库

首先，重点在于理解为何要考虑在组件中使用状态，以及它为什么能够提供多种帮助。

大部分 React 教程或者构建模板都包括了管理应用状态的外部库，如 Redux 或 MobX。

这就造成了一种普遍误解，即只靠 React 无法写出有状态的应用，而事实远非如此。

最明显的后果就是许多开发者尝试同时学习 React 和 Redux，以至于他们从未弄清楚如何正确使用 React 状态。

本节的目的就是弄清楚如何正确使用状态，并理解为何某些情况下不需要任何外部库。

3.2.2　工作原理

除了工厂方法和继承 `Component` 声明初始状态的方式不同，我们学习的另一个重要概念是，每个有状态的 React 应用都可以拥有初始状态。

在组件的生命周期中，可以使用生命周期方法或者事件处理器中的 `setState` 多次修改状态。当状态发生变化时，React 就用新状态渲染组件，这也是文档经常提到 React 组件类似状态机的原因。

用新状态（或者其中一部分）调用 `setState` 方法时，对象会合并到当前状态上。举例来说，假设初始状态如下所示：

```
this.state = {
  text: 'Click me!',
}
```

接着用新参数调用 `setState`：

```
this.setState({
  cliked: true,
})
```

最终的状态如下所示：

```
{
  cliked: true,
  text: 'Click me!',
}
```

当状态发生改变时，React 会再次执行渲染方法，因此除了设置新状态，我们不用做任何事。

然而某些情况下可能需要在状态更新完成时执行一些操作，React 为此提供了一个回调函数：

```
this.setState({
  clicked: true,
}, () => {
  console.log('the state is now', this.state)
})
```

将任意函数作为 setState 的第二个参数传递，状态更新完成时会触发该函数，同时组件完成渲染。

3.2.3 异步

应该总是将 setState 方法当作异步的，因为官方文档的介绍如下所示：

> 无法确保调用 setState 的同步操作[……]

实际上，如果在事件处理器中触发了 setState 后，尝试将当前状态值打印到控制台中，那么获得的是旧状态值：

```
handleClick() {
  this.setState({
    clicked: true,
  })
  console.log('the state is now', this.state)
}

render() {
  return <button onClick={this.handleClick}>Click me!</button>
}
```

以上述代码段为例，控制台上将会输出 the state is now null。发生这种情况的原因在于 React 知道如何优化事件处理器内部的状态更新，并进行批处理，以获得更好的性能。

如果稍微修改一下代码：

```
handleClick() {
  setTimeout(() => {
    this.setState({
      clicked: true,
    })

    console.log('the state is now', this.state)
  })
}
```

结果将是：

```
the state is now Object {clicked: true}
```

这与我们一开始预料的一样，因为 React 无法优化执行过程，只能尝试尽快更新状态。

注意，示例使用 setTimeout 只是为了展示 React 的行为，你永远不要这样编写事件监听器。

3.2.4 React lumberjack

前文提过，React 的工作方式很像状态机，每当状态改变就重新渲染。得益于这个特点，我们可以应用或撤销状态变化，并在整个过程中前进或者后退，这对调试很有帮助。

react-lumberjack 库对于理解以上内容相当有用。它的作者 Ryan Florence 参与开发了最流行的 React 库之一：react-router。

react-lumberjack 的使用非常简单，不过要记得在生产环境中禁用它。可以像任何 npm 包一样安装并导入，也可以直接按以下方式从 https://unpkg.com 引用它。

```
<script src="https://unpkg.com/react-lumberjack@1.0.0"></script>
```

脚本加载完成后，只需要使用应用让组件修改自身的状态即可。

如果某个地方出错或者想要调试应用的某个特殊状态，可以打开控制台并输入以下代码：

```
Lumberjack.back()
```

上述代码可以在时间上回退并撤销状态的改变，再查看以下代码：

```
Lumberjack.forward()
```

上述代码可以在时间上前进并重新应用状态的改变。

这个库处于试验阶段，不远的将来可能会消失，也可能成为 React 开发者工具的一部分，我们提到它是为了向你展示状态工作原理的实例。

3.2.5 使用状态

现在我们已经知道了状态的工作原理，接下来需要理解其使用时机以及何时应该避免在状态中保存值。

如果遵循规则，那么就能轻易搞清楚组件设计成无状态或有状态的时机，以及如何处理状态，以便可以在整个应用中复用组件。

首先，应该牢记只能将满足需求的最少数据放到状态中。

举例来说，如果要在点击按钮时改变标签，那么此时不应该保存标签文本，只需要保存布尔标记来表示是否已经点击按钮。

这样就是正确使用了状态，我们可以始终根据布尔标记重新计算不同的值。

其次，触发事件时只应将需要更新的值添加到状态中，然后重新渲染组件。

isClicked 标记和提交前的输入框的值都是很好的示例。

总的来说，应该只将记录当前 UI 所需的信息保存到状态中，如标签菜单的当前选中项。

判断状态是否适合保存信息的另一种方式是检查组件外部或子组件是否需要我们所维护的数据。

如果多个组件都需要跟踪同一份信息，那么应该考虑使用应用层级的状态管理器，如 Redux。

接下来我们将看看哪些情况下应该避免使用状态，以遵循最佳实践指南。

1. 可派生的值

只要能根据 props 计算最终值，就不应该将任何数据保存在状态中。

例如，如果从 props 接收了货币单位及价格，且总要一同展示它们，那么我们可能会认为将它保存在状态中更好，并在渲染方法内使用状态值，如下所示：

```
class Price extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      price: `${props.currency}${props.value}`
    }
  }

  render() {
    return <div>{this.state.price}</div>
  }
}
```

如果在父组件中按照以下方式创建，那么这种做法是可行的：

```
<Price currency="£" value="100" />
```

问题在于，如果货币单位或价格在 `Price` 组件的生命周期内发生改变，则永远不会重新计算状态（因为只会调用构造器一次），应用就会显示错误的价格。

因此，只要可以，就应该用 props 来计算值。

参考前面章节的做法，可以直接在渲染方法中使用一个辅助函数：

```
getPrice() {
  return `${this.props.currency}${this.props.value}`
}
```

2. 渲染方法

始终牢记，设置状态会触发组件重新渲染。因此，应该只将渲染方法要用到的值保存在状态中。

举例来说，如果需要保存组件要用到的 API 订阅或超时变量，而这些数据又不会影响渲染过程，那么应该考虑将它们放入独立模块。

以下代码的做法是错误的，因为之后要用的值位于状态中，但渲染方法又没有用到，这会在设置新状态时触发一次不必要的渲染：

```
componentDidMount() {
  this.setState({
    request: API.get(...)
  })
}

componentWillUnmount() {
  this.state.request.abort()
}
```

就上述场景而言，更好的做法是将 API 请求保存在外部模块中。

另一种常见做法是将请求保存为组件实例的私有成员：

```
componentDidMount() {
  this.request = API.get(...)
}

componentWillUnmount() {
  this.request.abort()
}
```

这种做法将请求封装到组件内部，但不会影响状态，因此，值发生改变时也就不会触发任何额外渲染。

Dan Abramov 创建了一张速记图来帮助我们做出正确选择，如下所示：

```
function shouldIKeepSomethingInReactState() {
  if (canICalculateItFromProps()) {
    // 不要把props里的数据复制到状态里，
    // 直接在render()方法内计算即可。
    return false;
  }
  if (!amIUsingItInRenderMethod()) {
    // 不要把没有参与渲染的数据放进状态。
    // 比如API订阅的部分就适合放在外部
    // 模块的自定义私有字段或变量里。
    return false;
  }
  // 除了以上的情况，都可以使用状态！
  return true;
}
```

3.3　prop 类型

我们的目的是开发真正可复用的组件，为了实现这一目的，需要尽可能清晰地定义组件接口。

如果希望整个应用可以复用组件，关键要确保清晰地定义组件及其参数，以便能够直观使用。

React 提供了一个可以非常简单地表达组件接口的强大工具，只要提供组件期望接收的 prop 名称与对应的验证规则即可。

与属性类型相关的规则也包含该属性为必选还是可选，还提供了用于编写自定义验证函数的选项。

查看以下简单示例：

```
const Button = ({ text }) => <button>{text}</button>

Button.propTypes = {
  text: React.PropTypes.string,
}
```

以上代码段创建了一个无状态函数式组件，以接收一个类型为字符串的文本 prop。

非常好，这样一来，需要用到该组件的每个开发人员都知道如何正确使用了。

然而，有时仅添加属性还不够，因为这无法告知我们没有该属性时组件能否正常工作。

例如，没有文本的情况下，按钮组件无法正常操作，解决方法就是将该 prop 标记为必需：

```
Button.propTypes = {
  text: React.PropTypes.string.isRequired,
}
```

如果某个开发人员在另一个组件中使用了按钮组件，却没有设置文本属性，那么浏览器控制台就会给出以下警告：

```
Failed prop type: Required prop `text` was not specified
in `Button`.
```

需要强调的是，这种警告只会在开发模式下出现。生产版本的 React 出于性能原因禁用了 `propTypes` 验证。

React 提供了多种开箱即用的验证器：从数组到数字类型，再到组件类型。

它还提供了 `oneOf` 这样的工具函数，以接受对某个特定属性有效的类型数组。

记住，我们应该始终将基本类型的 prop 传给组件，因为它们更容易验证和比较（第 10 章将介绍这种做法的优势）。

传递单一基本类型的 prop 有助于判断组件接口是否过泛，以及是否应该对其进行拆分。

如果意识到某个组件声明了太多 prop，而且它们之间没有关联，更好的做法是将组件纵向拆分为多个组件，然后每个组件附带少量的 prop 和职责。

然而某些情况下不可避免地要传递对象，此时需要用模型来定义 `propType`。

模型函数允许我们声明包含嵌套属性的对象，并为每个属性定义类型。

举例来说，如果要创建 `Profile` 组件，且该组件需要传入用户对象，其中包括必需的名字属性以及可选的姓氏属性，则可以按照以下方式定义：

```
const Profile = ({ user }) =>(
  <div>{user.name} {user.surname}</div>
)

Profile.propTypes = {
  user: React.PropTypes.shape({
    name: React.PropTypes.string.isRequired,
    surname: React.PropTypes.string,
  }).isRequired,
}
```

如果 React 现有的 `propTypes` 无法满足需求，那么我们可以创建自定义函数来验证属性：

```
user: React.PropTypes.shape({
  age: (props, propName) => {
    if (!(props[propName] > 0 && props[propName] < 100)) {
      return new Error(`${propName} must be between 1 and 99`)
    }
    return null
  },
})
```

例如，上述代码段验证了年龄字段是否属于特定区间；如果不属于，则返回错误。

React Docgen

得益于 prop 类型，组件的边界已经定义得很清晰了，我们还可以进行另一个操作，以便它们更易使用和共享。

诚然，如果 prop 类型的名称与类型都很清晰，开发人员应该就能充分利用它们，但我们可以做得更好。

可以从 prop 类型的定义起步，自动为组件生成文档。

`react-docgen` 库可以实现这个目的，执行以下命令来安装这个库：

```
npm install --global react-docgen
```

React Docgen 会读取组件的源代码，并从 prop 类型及其注释中提取相关信息。

回到我们最初创建的按钮组件示例：

```
const Button = ({ text }) => <button>{text}</button>

Button.propTypes = {
  text: React.PropTypes.string,
}
```

接着执行以下代码：

```
react-docgen button.js
```

最后会得到以下对象：

```
{
    "description": "",
    "methods": [],
    "props": {
        "text": {
            "type": {
                "name": "string"
            },
            "required": false,
            "description": ""
        }
    }
}
```

以上的 JSON 对象表示组件的接口。其中的 props 属性包括了类型为字符串的文本属性。

接着查看添加注释后的情况：

```
/**
 * A generic button with text.
 */
const Button = ({ text }) => <button>{text}</button>

Button.propTypes = {
  /**
   * The text of the button.
   */
  text: React.PropTypes.string,
}
```

再次执行命令会得到以下结果：

```
{
    "description": "A generic button with text.",
    "methods": [],
    "props": {
        "text": {
            "type": {
```

```
                "name": "string"
            },
            "required": false,
            "description": "The text of the button."
        }
    }
}
```

现在可以利用返回的对象来创建文档，并与团队成员共享或发布到 GitHub。

输出结果为 JSON 实际上使得这项工具变得非常灵活，因为用 JSON 对象填充模板后很容易生成网页。

用 docgen 提供组件文档的用例可以参见优秀的 Material UI 库，其所有文档都是根据源代码自动生成的。

3.4 可复用组件

我们已经了解了创建组件的最佳方式及应该使用本地状态的场景。我们还学习了如何用 prop 类型定义清晰的接口，以便组件可以复用。

现在我们来看一个真实示例，研究如何将一个不可复用的组件改成接口清晰通用的可复用组件。

假设组件从 API 路径加载一个消息集合，并在屏幕上显示列表。

这个示例很简单，但对于理解使得组件可复用的必要步骤很有用。

组件的定义方式如下所示：

```
class PostList extends React.Component
```

其中包括构造器和一个生命周期方法：

```
constructor(props) {
  super(props)

  this.state = {
    posts: [],
  }
}

componentDidMount() {
  Posts.fetch().then(posts => {
    this.setState({ posts })
  })
}
```

一个空数组被赋给消息，以表示初始状态。

调用 componentDidMount 时会触发 API 调用，取回的数据将保存到状态中。

这种数据获取模式相当常见，第 5 章将介绍更多可用方式。

辅助类 Posts 用来与 API 通信，它的获取方法会返回一个 Promise 对象，请求成功后会返回消息列表。

现在我们来看看显示消息列表部分的代码：

```
render() {
  return (
    <ul>
      {this.state.posts.map(post => (
        <li key={post.id}>
          <h1>{post.title}</h1>
          {post.excerpt && <p>{post.excerpt}</p>}
        </li>
      ))}
    </ul>
  )
}
```

我们在 render 方法内遍历了消息，并将其中的每条消息都映射为<li>元素。

假设始终需要显示标题字段，并将它包裹在<h1>标签中，而摘要属性是可选的，如果存在，就显示在段落中。

上述组件可以正常工作，而且没有任何问题。

现在，假设我们需要渲染一个类似的列表，不过这次想要显示的是从 props 而不是状态中获取的用户列表（以明确表示我们能应对不同场景）：

```
const UserList = ({ users }) => (
  <ul>
    {users.map(user => (
      <li key={user.id}>
        <h1>{user.username}</h1>
        {user.bio && <p>{user.bio}</p>}
      </li>
    ))}
  </ul>
)
```

传入用户集合，上述代码会渲染与消息示例类似的无序列表。

不同之处在于标题（heading），本例中的标题是用户名，而非之前的消息标题；还有可选部分要换成用户的简历属性，如果存在，就显示出来。

复制代码往往不是最佳解决方案，因此我们来看看 React 如何帮助代码符合 DRY（don't repeat yourself，不要重复自己）原则。第一步先创建可复用的列表组件，通过定义通用的集合属性，对该列表组件做一些抽象并与显示的数据解耦。最主要的需求在于，消息列表要显示标题和摘要属性，而用户列表要显示用户名和简历属性。

为了实现这个需求，我们创建两个 prop：`titleKey` 用于指定需要显示的属性名，`textKey` 则用于指定可选部分。

可复用的新 `List` 的 prop 如下所示：

```
List.propTypes = {
  collection: React.PropTypes.array,
  textKey: React.PropTypes.string,
  titleKey: React.PropTypes.string,
}
```

由于 `List` 组件不会包含任何状态或函数，可以将其写为无状态函数式组件：

```
const List = ({ collection, textKey, titleKey }) => (
  <ul>
    {collection.map(item =>
      <Item
        key={item.id}
        text={item[textKey]}
        title={item[titleKey]}
      />
    )}
  </ul>
)
```

`List` 组件接收 prop，并对集合进行迭代，将所有数据项映射为（将要创建的）另一个组件。如你所见，子组件传入了标题和文本这两个 prop，分别表示主属性和可选属性的值。

`Item` 组件非常简洁：

```
const Item = ({ text, title }) => (
  <li>
    <h1>{title}</h1>
    {text && <p>{text}</p>}
  </li>
)

Item.propTypes = {
  text: React.PropTypes.string,
  title: React.PropTypes.string,
}
```

至此，我们创建了两个接口清晰的组件，可以用它们来显示消息、用户以及任何其他类型的列表。小型组件有很多优点：它们更易维护与测试，bug 的定位与修复也更方便。

非常好，现在可以重写 PostList 和 UserList 组件了，以便这两个组件可以使用通用的可复用列表并能够避免代码重复。

修改 PostList 组件的渲染方法，如下所示：

```
render() {
  return (
    <List
      collection={this.state.posts}
      textKey="excerpt"
      titleKey="title"
    />
  )
}
```

UserList 组件同理：

```
const UserList = ({ users }) => (
  <List
    collection={users}
    textKey="bio"
    titleKey="username"
  />
)
```

我们用 prop 创建通用清晰的接口，使得一个面向单一需求的组件变得可复用。

现在可以在应用中多次复用这个组件了。有了 prop 类型的帮助后，每个开发人员都能轻易理解它的实现。

再进一步，可以用 react-docgen 为这个可复用列表生成文档，具体做法参考前文。

用可复用组件代替与数据耦合的组件好处非常多。

举例来说，假设我们想要添加新的逻辑，以实现点击按钮时隐藏或显示可选区域；或者新的需求是检查标题属性是否超过 25 个字符，超过就要截断并加上连字符。

此时我们只需要修改一处代码，用到该组件的所有组件就都能获得本次修改成果。

3.5　可用的风格指南

创建 API 清晰的可复用组件能很好地在应用内避免代码重复，但这可不是可复用性值得关注的唯一理由。

实际上，创建接受清晰的 prop 并与数据解耦的简洁组件是与团队其他成员共享基础组件库的最佳方式。基础通用且可复用的组件可以作为开箱即用组件，你可以将它们共享给团队中的其

他开发人员或者设计师。

例如，我们在前文创建了标题与摘要文本的通用列表，正因为与所显示的数据解耦，所以可以在应用中多次使用它，只要传入正确的 prop 即可。如果必须实现新的分类列表，只要将分类集合传递给列表组件就可以了。

问题在于，新的开发人员有时很难确定某些组件是否已经存在或者需要新增。解决方案通常是创建一套风格指南；这是一个非常强大且高效的工具，可以在团队内共享一套元素。

风格指南收集了可以跨页面使用的每个应用组件，并提供视觉展示。它非常有用，可以在拥有不同技能的团队成员间交换信息，并随着时间及组件数量的增加保持风格一致。

遗憾的是，创建 Web 应用的风格指南没那么容易，因为问题往往不够明确，而且实现需求上的细微差别要重复实现某些元素。React 为此提供了很大帮助，它创建了定义清晰的组件，并提供了一套风格指南，因此不需要我们再花什么精力。

不是只有 React 可以使得开发可复用组件变得更简单，其他工具也能帮助我们按照组件自身的代码来构建一套视觉展示库，其中之一就是 `react-storybook`。

React Storybook 分离了组件，因此无须运行整个应用就能渲染单个组件，这对开发和测试来说都非常完美。

正如名字所描述的那样，React Storybook 允许你编写故事文档来表示组件的可能状态。举例来说，如果要创建一个待办事项列表，可以编写故事文档来表示选中事项，再用另一个故事文档来描述未选中事项。

这个工具在跨团队共享组件方面非常强大，还能改进与其他开发人员的合作。只要查看已有的故事文档，刚加入公司的开发人员就能弄清是否需要创建新组件，或者是否已经有组件可以解决某个特定问题。

我们将 Storybook 应用到前面章节中的 `List` 组件示例。首先需要安装这个库：

```
npm install --save @kadira/react-storybook-addon
```

安装完成后，就可以开始编写故事文档。

列表项包含必需的标题属性和可选的文本属性，因此至少需要编写两条文档来表示相应状态。

故事文档通常放在名为 stories 的文件夹中，这个文件夹可以位于组件文件夹下或者文件目录中任何合适的地方。

你可以在 stories 文件夹下为每个组件创建一个文件。

本示例将在 list.js 中定义故事文档。

首先，从库中导入主要函数：

```
import { storiesOf } from '@kadira/storybook'
```

接着用此函数来定义故事文档，如下所示：

```
storiesOf('List', module)
  .add('without text field', () => (
    <List collection={posts} titleKey="title" />
  ))
```

可以用 `storiesOf` 函数定义组件名，并添加相应的故事文档，每条文档包括一段描述以及一个函数，该函数必须返回将要渲染的组件。

假设 `posts` 是与 React 相关的博客消息集合，如下所示：

```
const posts = [
  {
    id: 1,
    title: 'Create Apps with No Configuration',
  },
  {
    id: 2,
    title: 'Mixins Considered Harmful',
  },
]
```

运行 Storybook 并查看组件的视觉展示前，需要进行配置。

先在应用的根文件夹下创建.storybook 文件夹。

然后在.storybook 文件夹下创建 config.js 文件来加载故事文档：

```
import { configure } from '@kadira/storybook'

function loadStories() {
  require('../src/stories/list')
}

configure(loadStories, module)
```

从库中导入配置函数，然后定义另一个函数按照每条故事文档的路径加载它们。

接着将该函数传给配置函数。至此，一切就绪了。

最后需要创建 `npm` 任务，触发 Storybook 的可执行命令来运行它，然后在浏览器中查看风格指南。

具体做法如下所示：

```
"storybook": "start-storybook -p 9001"
```

将这条命令写在 package.json 的脚本部分。

现在只需要运行：

```
npm run storybook
```

并在浏览器中打开 http://localhost:9001。

页面左侧的故事文档列表可以访问 Storybook 的接口。

点击任何一条故事文档，就能在右侧看到对应的组件被渲染。

非常好，现在我们有了一套能为所有组件状态提供文档的可用风格指南，更易与设计师及产品经理共享信息。

最后再来创建第二条故事文档。

列表可以显示各项的标题和文本，因此在消息集合中加入第二个属性：

```
const posts = [
  {
    id: 1,
    title: 'Create Apps with No Configuration',
    excerpt: 'Create React App is a new officially supported...',
  },
  {
    id: 2,
    title: 'Mixins Considered Harmful',
    excerpt: '"How do I share the code between several...',
  },
]
```

将以下代码段编写的文档添加到刚刚那条文档后面：

```
.add('with text field', () => (
  <List collection={posts} titleKey="title" textKey="excerpt" />
))
```

现在回到浏览器，页面会自动刷新，接着就可以在左侧边栏中看到两条故事文档。

点击不同的文档，右侧的组件就会更新。

选择第一条文档，就可以看到只带有标题的列表；选择第二条，就会看到既有标题又有摘要的列表。

对于更复杂的组件，可以添加多条故事文档，并显示每个组件可能具有的所有状态与变体。

3.6　小结

学习如何编写可复用组件的旅程就要结束了。

我们一开始深入学习了基础知识，了解了有状态与无状态组件的差别，也研究了如何将紧密耦合的组件改成可复用的。接着我们学习了组件的内部状态，以及何时应该避免使用它。此外，我们还学习了 prop 类型的基础知识，并将这些概念应用到我们编写的可复用组件中。

最后，我们探讨了可用的风格指南如何帮助我们更好地与团队其他成员沟通，避免重复创建组件，并确保应用内的一致性。

接下来我们将学习组合组件时所能用到的各种技巧。

第 4 章 组合一切

我们在上一章中探讨了如何创建接口清晰的可复用组件。现在是时候学习如何让组件彼此高效通信了。

React 非常强大，因为它允许你组合可测试且可复用的小型组件来构建复杂的应用。你可以采用这种方式来控制应用的每一个单独部分。

本章将介绍一些最流行的组合模式与工具。

本章包含如下内容。

- 组件间如何通过 props 以及 `children` 进行通信。
- 容器组件与表现组件模式，以及该模式如何使得代码更易维护。
- mixin 试图解决的问题以及其失败的原因。
- 什么是高阶组件，以及如何用它更好地架构应用。
- `recompose` 库及其开箱即用的函数。
- 如何与上下文交互，并避免组件与它耦合。
- 什么是函数子组件模式，它有什么优势。

4.1 组件间的通信

复用函数是我们作为开发者的目标之一，而且我们也见识到了 React 可以很方便地创建可复用组件。

可以在应用的多个部分共享**可复用组件**，从而避免重复冗余。

接口清晰的小型组件可以组合出复杂的应用，同时又能确保应用的强大和可维护性。

React 组件的组合方式相当直观，将它们放入 `render` 方法即可：

```
const Profile = ({ user }) => (
  <div>
```

```
    <Picture profileImageUrl={user.profileImageUrl} />
    <UserName name={user.name} screenName={user.screenName} />
  </div>
)

Profile.propTypes = {
  user: React.PropTypes.object,
}
```

如上所示，`Picture` 组件用于显示个人资料照片，`UserName` 组件用于显示用户名及屏幕昵称，简单地组合两者就能创建 `Profile` 组件。

通过这种方式，只需要编写几行代码就可以很快生成 UI 新的部分。

正如以上示例所示，创建组件时会通过 props 在它们之间共享数据。

父组件通过 props 将数据向下传递，组件树中的每个组件都能接受这份数据（或者其中一部分）。

当一个组件向另一个组件传递某些 props 时，不论两者间是否存在父子关系，传出组件称为**拥有者**。

以前面的代码段为例，`Profile` 不是 `Picture` 的直接父组件（`div` 标签才是），但 `Profile` 拥有 `Picture`，因为前者向后者传递了 props。

children

`children` 是一个特殊的 prop，拥有者组件可以将它传递给渲染方法内定义的组件。

React 文档将 `children` 属性描述为**不透明的**，因为它没有对所包含的值提供任何说明。

父组件的渲染方法中定义的子组件通常接收 props 作为自身的 JSX 属性，或者作为 `createElement` 函数的第二个参数。

定义组件时也可以包含内部的嵌套组件，可以用 `children` 属性访问这些子组件。

查看以下的 `Button` 组件，其文本属性表示按钮的文本：

```
const Button = ({ text }) => (
  <button className="btn">{text}</button>
)

Button.propTypes = {
  text: React.PropTypes.string,
}
```

可以按以下方式使用该组件：

```
<Button text="Click me!" />
```

渲染的代码如下所示：

```
<button class="btn">Click me!</button>
```

现在，假设我们想要在应用的多个部分使用类名一样的相同按钮，并且我们不只是想要显示单个简单字符串。

实际的 UI 包含各种按钮，如文本按钮、文本图标按钮及文本标签按钮。

大部分情况下，较好的做法是为 `Button` 组件添加多个参数，或者创建不同版本的 `Button` 组件，每个版本具有自己独立的特性，如 `IconButton` 组件。

然而，如果只将 `Button` 组件看作一个封装器，并想要在其内部渲染任何元素，此时可以使用 `children` 属性。

这个目的很容易实现，只要将原先的 `Button` 组件改成类似以下的代码段即可：

```
const Button = ({ children }) => (
  <button className="btn">{children}</button>
)

Button.propTypes = {
  children: React.PropTypes.array,
}
```

修改完成后，`Button` 组件就不再局限于简单的单个文本属性了，现在我们可以将任何元素传递给它，然后在 `children` 属性的位置上渲染出来。

在以上示例中，`Button` 组件内部封装的任何元素都会渲染成类为 `btn` 的按钮元素的子元素。

举例来说，如果我们想要在按钮内渲染一张图片和一个 `span` 元素包裹的文本，可以参考以下代码段：

```
<Button>
  <img src="..." alt="..." />
  <span>Click me!</span>
</Button>
```

以上代码段在浏览器中渲染的结果如下所示：

```
<button className="btn">
  <img src="..." alt="..." />
  <span>Click me!</span>
</button>
```

这种便捷方式允许组件接收任何 `children` 元素，并将它们封装在预先定义好的父组件中。

现在可以为 `Button` 组件传递图片、标签甚至其他 React 组件了，它们会被渲染为 `Button` 的子元素。

在以上示例中，我们将 `children` 属性定义为数组，这意味着可以传入任何数量的元素作为该组件的子元素。

可以传入单个子元素，如下所示：

```
<Button>
  <span>Click me!</span>
</Button>
```

传入单个元素时会看到以下提示：

```
Failed prop type: Invalid prop `children` of type `object` supplied
to `Button`, expected `array`.
```

这是因为组件只有单个子元素时，出于性能方面的考虑，React 会优化元素的创建过程，避免分配数组。

解决这个警告提示很简单，设置 `children` 接受以下 prop 类型即可：

```
Button.propTypes = {
  children: React.PropTypes.oneOfType([
    React.PropTypes.array,
    React.PropTypes.element,
  ]),
}
```

4.2　容器组件与表现组件模式

第 3 章介绍了如何逐步将耦合的组件改为可复用组件。本节将探讨如何为组件应用一种类似的模式，以便它们更清晰，更易维护。

React 组件通常包含杂合在一起的**逻辑与表现**。逻辑一般指与 UI 无关的那些东西，如 API 的调用、数据操作以及事件处理器。表现则是指渲染方法中创建元素用来显示 UI 的部分。

React 有一种简洁而强大的模式，称为**容器组件与表现组件**，按照这种模式创建组件可以帮助我们分离上述两个关注点。

清晰地定义逻辑与表现间的界限不仅能使组件更易复用，还有很多其他好处，本节将一一介绍。

再次强调，学习新概念的最佳方式之一就是查看实际示例，因此我们来研究一些代码。

假设我们有一个组件利用地理位置 API 获取用户定位，并在浏览器页面上显示经纬度。

首先，在组件文件夹下创建 geolocation.js 文件，并用 `class` 定义 `Geolocation` 组件：

```
class Geolocation extends React.Component
```

接着定义 `constructor` 方法，用于初始化内部状态并绑定事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    latitude: null,
    longitude: null,
  }

  this.handleSuccess = this.handleSuccess.bind(this)
}
```


现在可以用 `componentDidMount` 回调触发 API 请求了：

```
componentDidMount() {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(this.handleSuccess)
  }
}
```

当浏览器返回数据时，使用以下函数将结果保存在状态中：

```
handleSuccess({ coords }) {
  this.setState({
    latitude: coords.latitude,
    longitude: coords.longitude,
  })
}
```

最后用 `render` 方法显示经纬度：

```
render() {
  return (
    <div>
      <div>Latitude: {this.state.latitude}</div>
      <div>Longitude: {this.state.longitude}</div>
    </div>
  )
}
```

值得一提的是，首次渲染时经纬度的数据都是 `null`，因为我们在组件挂载后才向浏览器请求坐标。在现实的组件中，可能会想要在数据返回前显示一个旋转动画。要实现这一需求，可以使用第 2 章中学到的某个条件语句技巧。

现在这个组件没有任何问题了，可以按预期工作。

假设现在你正和设计师一起探讨组件的 UI 部分，即向用户展示经纬度信息的地方。

为了更快地迭代，将 UI 与请求并加载位置信息的部分分开会不会更好？

抽离主组件的表现部分，可以使用 Storybook 在**风格指南**中用伪数据渲染组件，像上一章那样，这样就能享受到开发可复用组件带来的所有好处。

那么我们来看看如何通过遵循容器组件与表现组件模式来实现这个目的。

在这个模式中，每个组件都拆分成两个小组件，每个小组件各自都有清晰的职责。

容器组件包含有关组件逻辑的一切，API 的调用就在容器组件中进行。此外，它还负责处理数据操作以及事件处理。

UI 定义在表现组件中，并且表现组件以 prop 的形式从容器组件接收数据。

因为表现组件通常不含逻辑，所以可以将它创建为函数式无状态组件。

没有规则要求表现组件一定不能拥有状态。例如，UI 状态就可以保存在表现组件内部。

这个示例只需要一个用于显示经纬度的组件，因此用一个简单函数来实现即可。

首先，将 `Geolocation` 组件重命名为 `GeolocationContainer`：

```
class GeolocationContainer extends React.Component
```

同时将 geolocation.js 文件重命名为 geolocation-container.js。

在容器组件名的末尾加上 container，而表现组件则采用原有名称，这项规则并不严格，却是 React 社区广泛使用的最佳实践。

另外还需要更改渲染方法的实现，并移除所有 UI 部分，如下所示：

```
render() {
  return (
    <Geolocation {...this.state} />
  )
}
```

如以上代码段所示，我们不在容器组件的渲染方法内创建 HTML 元素，而是使用（接下来将会创建的）表现组件，并传入状态。

状态包含经纬度属性，默认值为 `null`，浏览器触发**回调函数**后会将真正的用户位置信息赋值给它们。

这里用到了第 2 章中介绍的扩展属性操作符。这种方式可以很方便地传入状态的属性，无须手动逐个书写 prop。

现在我们创建一个名为 geolocation.js 的新文件，在该文件中定义无状态函数式组件，如下所示：

```
const Geolocation = ({ latitude, longitude }) => (
  <div>
    <div>Latitude: {latitude}</div>
    <div>Longitude: {longitude}</div>
  </div>
)
```

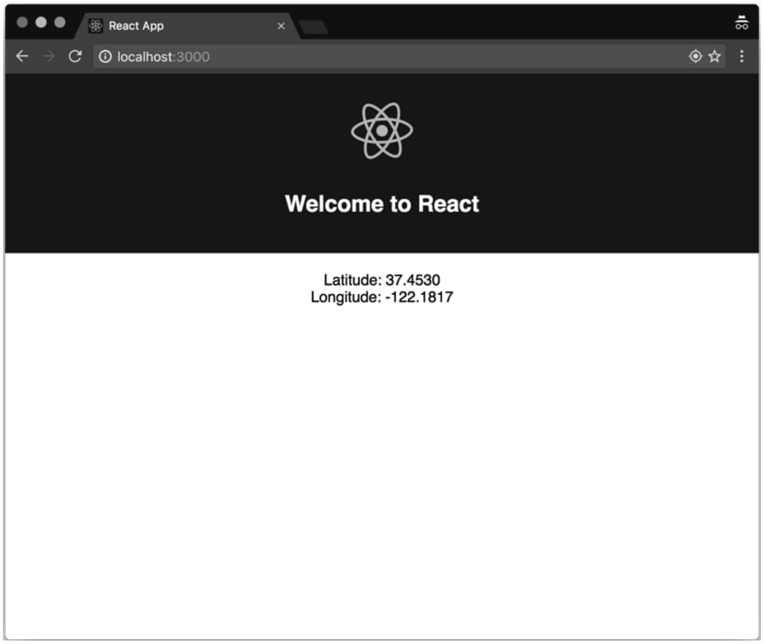
无状态函数式组件就是纯粹函数，传入状态并返回元素，可以非常优雅地定义 UI。

在这个示例中，函数从拥有者组件那接收经纬度数据，然后返回标记结构来显示出来。

我们当然希望能遵循最佳实践，为组件定义清晰的接口，因此用 `propTypes` 来声明组件所需要的属性：

```
Geolocation.propTypes = {
  latitude: React.PropTypes.number,
  longitude: React.PropTypes.number,
}
```

如果在浏览器中运行组件代码，那么可以看到如下图所示的内容：



通过遵循容器组件与表现组件模式，我们创建了一个**不含数据获取逻辑**[1]的可复用组件，可以将该组件放入风格指南，并传入模拟的坐标数据。

① 原文为 dumb，中文表示哑，此处表示组件不能获取数据。——译者注

如果应用的其他部分需要显示同样的数据结构，不必创建新的组件；只要将现有组件封装进新的容器组件即可，这个新容器会从不同的 API 路径加载经纬度数据。

与此同时，团队的其他成员可以添加错误处理逻辑来改进这个地理位置容器组件，而且不会影响表现组件。

其他人甚至可以临时开发一个表现组件，只用于显示与调试数据，当数据准备就绪时再替换成真正的表现组件。

能够并行开发同一个组件对于整个团队来说收益很大，尤其是那些迭代开发接口的公司。

这个模式简单却又非常强大，在大型应用中使用它可以为开发速度及项目可维护性带来巨大影响。

另一方面，如非真正需要却使用了这种模式会带来反面问题，代码库需要创建更多文件及组件，进而导致可用程度降低。

因此，当决定按照容器组件与表现组件模式进行重构时，应当仔细思考一番。

总的来说，遵循该模式的正确途径是从单个组件着手，并且只在逻辑与表现过于紧密耦合时进行拆分。

拿以上示例来说，先从单个组件开始，后来我们才意识到可以从标记中分离 API 调用逻辑。

判断容器组件和表现组件分别要包含哪些内容往往不太直观；以下几点建议可以帮你做出判断。

容器组件：

- 更关心行为部分；
- 负责渲染对应的表现组件；
- 发起 API 请求并操作数据；
- 定义事件处理器；
- 写作类的形式。

表现组件：

- 更关心视觉表现；
- 负责渲染 HTML 标记（或其他组件）；
- 以 props 的形式从父组件接收数据；
- 通常写作无状态函数式组件。

4.3 mixin

组件对于实现可复用性意义重大，但不同场景下的不同组件拥有相同行为时怎么办？

我们不希望在应用中复制代码，当需要在不同组件间共享功能时，可以使用 React 提供的一个工具：mixin。

现在已经不再推荐使用 mixin 了，不过理解这项技术尝试解决的问题还是很有意义的，顺便了解一下有哪些可能的替代方案。

另外，你可能需要维护旧版 React 开发的遗留项目，此时了解 mixin 及其用法就很有意义了。

首先，mixin 只能和 `createClass` 工厂方法搭配使用，因此，如果你用的是类，那么就不能使用 mixin，这也正是不推荐使用它们的原因之一。

假设在应用中使用了 `createClass` 方法，你会发现需要在不同组件内编写相同的代码。

举例来说，你需要监听 window 的 resize 事件来获取窗口大小，并执行相应操作。

mixin 的一种用法是一次编写，然后在不同组件中共享。我们来探究一段代码。

mixin 可以定义为对象字面量，和组件拥有同样的方法与属性：

```
const WindowResize = {...}
```

mixin 通常用状态与组件进行通信。通过 `getInitialState` 就能用 window 的初始 `innerWidth` 对状态进行初始化：

```
getInitialState() {
  return {
    innerWidth: window.innerWidth,
  }
},
```

现在我们想要追踪值的变化。因此，当组件挂载时，开始监听 window 的 resize 事件：

```
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
},
```

我们还想在组件完成卸载后立马移除事件监听器。这非常关键，可以释放内存，避免在 window 上留下无用的监听器：

```
componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
},
```

最后，定义每次触发 window resize 事件时的回调函数。

实现回调函数，以便根据新的 innerWidth 值来更新状态，这样一来，使用该 mixin 的组件就会用新的值重新渲染自身：

```
handleResize() {
  this.setState({
    innerWidth: window.innerWidth,
  })
},
```

从以上代码段可以看出，mixin 的创建方式和组件很像。

现在，如果想要在组件中使用 mixin，只需要将它放入对象的 mixin 数组属性中：

```
const MyComponent = React.createClass({

  mixins: [WindowResize],

  render() {
    console.log('window.innerWidth', this.state.innerWidth)
    ...
  },

})
```

从此刻起，可以在组件的状态中获取 window 的 innerWidth 值了，只要 innerWidth 发生变化，组件就会用更新后的值进行重新渲染。

可以同时在多个组件中使用这个 mixin，也可以在一个组件中使用多个 mixin。

mixin 具有一项很棒的特性，这个特性允许它们合并生命周期方法和初始状态。

举例来说，如果在某个组件中使用了 WindowResize mixin，同时该组件也定义了 componentDidMount 钩子，那么二者会按顺序执行。

使用相同生命周期钩子的多个 mixin 同理。

现在我们来了解一下 mixin 存在的问题，下一节将介绍实现相同结果的最佳技巧，同时还能避免一切问题。

首先，mixin 有时利用内部函数与组件进行通信。

例如，WindowResize 这个 mixin 可能希望组件实现 handleResize 函数，并在窗口尺寸变化时允许开发者自由地执行某些操作，而不是用状态来触发更新。或者，相较于在状态中设置新值，mixin 可能需要组件调用一个函数（如示例中的 getInnerWidth）来获取实际值。

问题是，我们无法得知需要实现哪些方法。

这对于可维护性来说尤为糟糕，因为如果使用了多个 mixin，那么组件最终需要实现不同的

方法，当移除某些 mixin 或者行为发生改变时，很难消除废弃代码。

mixin 的另一个常见问题是**冲突**。实际上，虽然 React 确实可以聪明地合并生命周期回调，但如果两个 mixin 定义或调用了同样的函数名，抑或在状态中使用了相同的属性，那么 React 对此无能为力。

这种情况对大型代码库来说非常糟糕，因为这会带来无法预料的行为，并且增加了调试问题的难度。

正如我们在 `WindowResize` 示例中所见，mixin 往往需要用状态与组件进行通信。因此，如果 mixin 在组件状态中更新了一个特殊属性，那么组件就会因为这个新属性重新进行渲染。

这会导致组件包含不必要的状态，这种做法并不好，因为我们已经知道，为了提升可复用性与可维护性，应该极力避免这样做。

最后，有时会出现 mixin 互相依赖的情况。例如，我们可以创建 ResponsiveMixin，它会根据窗口大小改变某些组件的可见度，而 `WindowResize` mixin 刚好提供了窗口大小值。

mixin 间的这种耦合导致组件重构和应用扩展变得非常困难。

4.4　高阶组件

4.3 节介绍了 mixin 在组件间共享功能方面的用处及其带来的问题。

第 2 章介绍函数式编程时，我们提到了**高阶函数**的概念，这类函数对传入的函数进行增强，并返回一个添加了额外行为的新函数。

我们来看看能否在 React 组件上应用相同概念，借此实现组件间共享功能，并避开 mixin 的缺点。

当高阶函数概念应用在组件上时，我们将它简称为**高阶组件**。

首先我们来看看高阶组件长什么样：

```
const HoC = Component => EnhancedComponent
```

高阶组件其实就是函数，它接收组件作为参数，对组件进行增强后返回。

我们通过一个很简单的示例来理解增强后的组件长什么样。

假设出于某些原因，你需要为每个组件添加相同的 `className` 属性。可以选择在全部的渲染方法中为每个组件加上 `className` prop，也可以参考以下示例编写一个高阶组件：

```
const withClassName = Component => props => (
  <Component {...props} className="my-class" />
)
```

一开始理解以上代码会有些困难，我们试着理解一番。

我们声明了接受 `Component` 参数的 `withClassName` 函数，然后返回另一个函数。

返回的函数是一个无状态函数式组件，它接受 props 参数并渲染原来的组件。整个 props 对象会展开，然后和值为`"my-class"`的 `className` 属性一起传给组件。

高阶组件通常将组件上接收到的 props 对象展开，这样做的原因是尽量让它们更直观，并且只添加新的行为。

这个示例相当简单，用处也不是很大，但它可以让你更好地理解高阶组件的定义以及它们的样子。

现在我们来看看如何在组件中使用 `withClassName` 高阶组件。

首先，创建一个无状态函数式组件，它接收类名称并赋值给一个 `div` 标签：

```
const MyComponent = ({ className }) => (
  <div className={className} />
)

MyComponent.propTypes = {
  className: React.PropTypes.string,
}
```

我们不直接使用它，而是传递给高阶组件，如下所示：

```
const MyComponentWithClassName = withClassName(MyComponent)
```

通过将组件封装进 `withClassName` 函数，确保该组件可以接收 `className` 属性。

接下来我们将做一些更令人激动的事情，试着将 4.3 节中提到的 `WindowResize` mixin 转换为高阶组件函数，以便整个应用都能复用。

mixin 的原理很简单，它监听 window 的 resize 事件，将 window 的 `innerWidth` 属性更新到状态中。

该 mixin 的最大问题其实是利用组件状态来提供 `innerWidth` 值。

这种做法不好的原因是，它用外来属性污染了状态，这些属性可能会与组件自身用到的属性发生冲突。

首先，创建接受组件作为参数的函数：

```
const withInnerWidth = Component => (
  class extends React.Component { ... }
)
```

你可能已经注意到了高阶组件的命名方式。这种做法很常见，就是给为组件提供信息的高阶组件名称加上 `with` 前缀。

`withInnerWidth` 函数将返回组件类而不是无状态函数式组件，如上述示例所示，我们需要额外的函数与状态。

我们来看一下返回了什么样的类。

构造器中定义了初始状态，并且 `handleResize` 回调函数绑定了当前类。

```
constructor(props) {
  super(props)

  this.state = {
    innerWidth: window.innerWidth,
  }

  this.handleResize = this.handleResize.bind(this)
}
```

生命周期钩子和事件处理器与 mixin 的完全一样：

```
componentDidMount() {
  window.addEventListener('resize', this.handleResize)
}

componentWillUnmount() {
  window.removeEventListener('resize', this.handleResize)
}

handleResize() {
  this.setState({
    innerWidth: window.innerWidth,
  })
}
```

最后，原先的组件按以下方式来渲染：

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

你可能注意到了，此处和之前一样展开了 props，同时也展开了状态。

实际上，我们将 `innerWidth` 值保存在（高阶组件的）状态中来实现最初的行为，同时改用 props，不污染原先组件的状态。

我们在第 3 章中介绍了使用 props 能始终很好地确保可复用性。

现在，高阶组件的使用以及 `innerWidth` 值的获取都非常直观了。

创建一个无状态函数式组件，并期望传入 `innerWidth` 属性：

```
const MyComponent = ({ innerWidth }) => {
  console.log('window.innerWidth', innerWidth)
  ...
}

MyComponent.propTypes = {
  innerWidth: React.PropTypes.number,
}
```

然后用以下代码增强该组件：

```
const MyComponentWithInnerWidth = withInnerWidth(MyComponent)
```

这比用 mixin 具有很多优势：首先没有污染任何状态，其次不需要组件来实现任何方法。

这意味着组件和高阶组件没有耦合，可以在整个应用中复用它们。

再次强调，用 props 代替状态能分离组件与逻辑，这样一来，我们就可以在风格指南中使用组件，忽略任何复杂逻辑，只传入 props 即可。

在这种特殊情况下，可以为支持的各种 `innerWidth` 尺寸分别创建一个组件。

思考以下示例：

```
<MyComponent innerWidth={320} />
```

以及：

```
<MyComponent innerWidth={960} />
```

4.5　recompose

一旦熟悉高阶组件，就会意识到它们的强大之处，并希望能够对其加以充分利用。

`recompose` 是一个很流行的库，提供了许多有用的高阶组件，而且可以优雅地组合它们。

该库提供的高阶组件就是一些用于封装组件的小工具，可以从组件中抽离部分逻辑，使它们更简洁、可复用性更好。

假设组件从某个 API 接收一个用户对象，且该对象包含很多属性。

允许组件接收任何对象的做法不太好，因为这依赖于组件了解对象长什么样，最重要的是，

一旦对象发生改变，组件就会崩溃。

从父组件接收 props 的更好方式是将每个属性定义为基本类型。

我们用 Profile 组件来显示 username 和 age，如下所示：

```
const Profile = ({ user }) => (
  <div>
    <div>Username: {user.username}</div>
    <div>Age: {user.age}</div>
  </div>
)

Profile.propTypes = {
  user: React.PropTypes.object,
}
```


如果想要改变组件接口来接收单个 prop 而不是整个用户对象，可以用 recompose 提供的高阶组件 flattenProp 来实现。

我们来看一下具体做法。

首先，修改组件来单独声明每个属性，如下所示：

```
const Profile = ({ username, age }) => (
  <div>
    <div>Username: {username}</div>
    <div>Age: {age}</div>
  </div>
)

Profile.propTypes = {
  username: React.PropTypes.string,
  age: React.PropTypes.number,
}
```

然后用高阶组件进行增强：

```
const ProfileWithFlattenUser = flattenProp('user')(Profile)
```

你可能已经注意到了，此处高阶组件的用法稍有不同。实际上，这是一种函数式编程方式，有些高阶组件先通过偏函数用法接收参数。

它们的特征如下所示：

```
const HoC = args => Component => EnhancedComponent
```

我们要做的就是先调用高阶组件来创建一个函数，然后用它封装原有组件：

```
const withFlattenUser = flattenProp('user')
const ProfileWithFlattenUser = withFlattenUser(Profile)
```

非常好！现在假设出于某些原因，需要改变用户名属性，以便组件更加通用、可复用性更好。

这种情况下可以使用 recompose 库提供的 renameProp 并更新组件，如下所示：

```
const Profile = ({ name, age }) => (
  <div>
    <div>Name: {name}</div>
    <div>Age: {age}</div>
  </div>
)

Profile.propTypes = {
  name: React.PropTypes.string,
  age: React.PropTypes.number,
}
```

现在我们希望同时使用多个高阶组件：一个用于扁平化处理用户 prop，另一个用于重命名用户对象的单个 prop，不过串联使用函数的做法似乎不太好。

此时 recompose 库提供的 compose 函数就派上用场了。

实际上，可以将多个高阶组件传给该函数，最终会得到单个增强后的高阶组件：

```
const enhance = compose(
  flattenProp('user'),
  renameProp('username', 'name')
)
```

然后按照以下方式将它应用于原有组件：

```
const EnhancedProfile = enhance(Profile)
```

这种方式显然更方便、优雅。

有了 recompose 库后，我们不仅可以使用它提供的高阶组件，还可以将 compose 函数用在我们自己的高阶组件上，甚至结合使用都可以：

```
const enhance = compose(
  flattenProp('user'),
  renameProp('username', 'name'),
  withInnerWidth
)
```

如你所见，compose 函数非常强大，大大提升了代码可读性。

可以串联多个高阶组件以尽量保持组件简洁。

非常重要的另一点是，不要滥用高阶组件，因为每层抽象都会带来一些问题。拿本例来说，性能就是需要权衡的地方。

你可以这样想，每对组件进行一层高阶封装，就是在添加新的渲染函数、新的生命周期方法调用以及内存分配。

出于这个原因，需要仔细斟酌何时应该使用高阶组件，以及何时重构架构才是更好的做法。

context

高阶组件可以很方便地处理 context。

React 始终提供了 context 特性，虽然关于它的文档出现得较晚，但许多库中还是能见到它的应用。

文档依然建议谨慎使用 context，因为它仍处于试验阶段，未来可能会改变。

但在某些场景下，它是一项非常强大的工具，能够帮助我们在组件树中传递数据，无须用 props 逐级传递。

要想利用 context 的优势，同时避免其 API 与组件产生耦合，可以使用高阶组件。

高阶组件可以从 context 中获取数据，转换成 props 后再传递给组件。

在这种方式下，组件不知道 context 的存在，也就能轻易地复用到应用的各个部分。

另一方面，如果 context 的 API 未来发生变化，应用中唯一需要修改的部分就是高阶组件，因为组件本身与它解耦了，这能带来很大益处。

`recompose` 库提供了一个函数，使得 context 的使用变得简单易懂，接收 props 的过程也更加直观。我们来看看该函数的工作原理。

假设你有一个用来显示货币单位和价值的 `Price` 组件。

context 的最广泛用法就是将通用配置从根节点向下传递到叶节点，货币单位就是这些配置值之一。

我们从一个与 context 耦合的组件入手，用高阶组件模式逐步将它改写成可复用的：

```
const Price = ({ value }, { currency }) => (
  <div>{currency}{value}</div>
)

Price.propTypes = {
  value: React.PropTypes.number,
}

Price.contextTypes = {
  currency: React.PropTypes.string,
}
```

上述代码中的无状态函数式组件接收了 props 的值属性，且 context 的货币单位属性作为第二个参数。

我们同时为这两个值定义了 prop 类型与 context 类型。

如你所见，这个组件不能真正地复用，因为它需要父组件将货币单位定义为子组件的 context 类型才能工作。

举例来说，我们不能简单地传入模拟的货币单位属性作为 prop，然后在风格指南中使用。

首先，对组件进行修改，以便从 props 中获取两个值：

```
const Price = ({ currency, value }) => (
  <div>{currency}{value}</div>
)

Price.propTypes = {
  currency: React.PropTypes.string,
  value: React.PropTypes.number,
}
```

当然，不能直接用它替代前面的示例，因为没有父组件为它设置货币单位 prop。

我们要将它封装进高阶组件，然后将 context 上接收的值转换成 props。

可以用 `recompose` 库提供的 `getContext` 函数，也可以简单地从零开始编写一层自定义封装。

这里要再次用到偏函数写法对高阶组件进行特殊化处理，然后多次复用它：

```
const withCurrency = getContext({
  currency: React.PropTypes.string
})
```

接着将它应用于组件：

```
const PriceWithCurrency = withCurrency(Price)
```

现在可以用刚刚完成的 `Price` 组件替换旧组件了，它仍然可以正常运行，且不会与 context 发生耦合。

这种做法大有裨益，因为我们不需要修改父组件，还可以利用 context 特性且无须担心 API 未来会发生变化，而且 `Price` 组件也实现了可复用性。

实际上，我们可以将任意的货币单位和值传递给该组件，不需要特定父组件提供这些值。

4.6　函数子组件

React 社区目前对一种名为**函数子组件**的模式达成了共识。

流行库 react-motion 广泛运用了该模式，第 6 章将详细介绍这个库。

这种模式的主要概念是，不按组件的形式传递子组件，而是定义一个可以从父组件接收参数的函数。

我们来看一下此类函数长什么样子：

```
const FunctionAsChild = ({ children }) => children()

FunctionAsChild.propTypes = {
  children: React.PropTypes.func.isRequired,
}
```

如你所见，FunctionAsChild 组件拥有定义为函数的 children 属性，并且它没有按 JSX 表达式的形式使用，而是作为函数被调用。

上述组件的用法如下所示：

```
<FunctionAsChild>
  {() => <div>Hello, World!</div>}
</FunctionAsChild>
```

原理很简单：父组件的渲染方法触发了子函数，返回了 div 标签包裹的 Hello, World!文本，最后显示在屏幕上。

接着我们来探讨一个更有实际意义的示例，其中父组件传递一些参数给 children 函数。

创建一个子组件为函数的 Name 组件，并为该函数传入字符串 World：

```
const Name = ({ children }) => children('World')

Name.propTypes = {
  children: React.PropTypes.func.isRequired,
}
```

上述组件的用法如下所示：

```
<Name>
  {name => <div>Hello, {name}!</div>}
</Name>
```

这段代码也渲染出了 Hello, World!文本，不过这次是由父组件传递了名字属性。

现在这种模式的工作原理应该很清晰了，我们来看看它具有哪些优点。

首要优点是，可以像高阶组件那样封装组件，在运行时为它们传递变量而不是固定属性。

以下的 Fetch 组件就是很好的示例，它从某个 API 路径加载数据，然后返回给 children 函数：

```
<Fetch url="...">
  {data => <List data={data} />}
</Fetch>
```

其次，以这种方式组合组件不要求 `children` 函数使用预定义的 prop 名称。

使用组件的开发者可以自行决定函数接收的变量的名称。

这使得函数子组件方案更加灵活。

最后，封装器的可复用程度很高，因为它不关心子组件要接收什么，只期望传入一个函数。

由于这一点，按函数子组件模式编写的组件也可以用于应用的不同部分，托管各种各样的子组件。

4.7 小结

本章介绍了如何组合可复用组件，并让它们高效地通信。

props 可以使得组件彼此解耦，并创建定义清晰的接口。

接着我们了解了 React 中最有趣的一些组合模式。

第一种就是所谓的容器组件与表现组件模式，它帮助我们从表现层抽离逻辑，并创建拥有单一职责的特定组件。

我们还见识了 React 尝试用 mixin 解决组件共享功能的问题。遗憾的是，mixin 解决这个问题的同时又引发了其他问题，影响了应用的可维护性。

要想实现这个目的而不增加复杂度，其中一种方法就是使用高阶组件。它们就是函数，传入组件并返回增强的版本。

`recompose` 库提供了一些有用的高阶组件，可以配合自定义的高阶组件一起使用，这样可以使得组件在实现上尽量少包含逻辑。

我们还学习了如何利用高阶组件来处理 context，避免其与组件发生耦合。

最后，我们学习了函数子组件模式，通过这种模式来动态地组合组件。

现在是时候讨论一下数据获取以及单向数据流方面的知识了。

第 5 章

恰当地获取数据

本章旨在介绍 React 应用可以使用的多种数据获取模式。

要想找到最佳模式，需要清晰地认识数据在 React 组件树中的流动方式。

理解父子组件间的通信方式很重要，理解无关联的兄弟组件间共享数据的方式也很关键。

我们将看到一些有关数据获取的真实示例，并通过高阶组件将基本组件的结构写得更好。

最后，我们将探讨 `react-refetch` 这类现成的库如何通过提供获取数据所需的核心功能，帮我们节省大量时间。

本章包含如下内容。

- React 的单向数据流，以及它如何使应用结构更易于理解。
- 子组件如何用回调函数与父组件通信。
- 两个兄弟组件如何通过公有父组件通信。
- 如何创建通用的高阶组件来获取任意 API 路径的数据。
- `react-refetch` 的工作原理，以及为何可以将这个有用的工具集成到项目中，从而更便捷地获取数据。

5.1 数据流

第 3~4 章介绍了如何创建单个可复用组件，以及如何有效地组合它们。

现在我们开始学习如何构建恰当的数据流，以便应用能够跨组件共享数据。

React 推行了一种非常有趣的模式，允许数据从根节点流向叶节点。这种模式通常称作**单向数据流**，本节将对其进行详细介绍。

顾名思义，React 中的数据流是单向的，即从组件树的顶部流向底部。这种方式有很多好处，它简化了组件行为以及组件间的关系，增强了代码的可预测性和可维护性。

每个组件都以 prop 的形式从父组件接收数据，并且 prop 无法修改。获取数据后，可以将其转换为新的形式或者往下传给其他子组件。每个子组件都能保存内部状态，也可以将状态作为自身嵌套组件的 prop。

到目前为止，我们见到的所有示例都是父组件通过 prop 将数据共享给子组件。

但是，如果子组件将数据向上传给父组件，会发生什么呢？当子组件的状态改变时，如何更新父组件呢？如果两个兄弟组件需要共享数据，应该怎么做？我们将通过一个真实示例来解答上述所有问题。

先从无子组件的简单组件起步，然后逐步将其改写得更清晰、结构更好。

这种方式将为我们展示每一阶段的最佳模式，以便应用于整棵树的数据流。

我们来探讨 `Counter` 组件的代码，该组件从 `0` 开始计数且拥有两个按钮，分别用于计数器值的增和减。

首先，创建一个继承 React 的 `Component` 函数的类：

```
class Counter extends React.Component
```

该类在构造器中将计数器初始化为 `0`，并在组件自身上绑定了事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    counter: 0,
  }

  this.handleDecrement = this.handleDecrement.bind(this)
  this.handleIncrement = this.handleIncrement.bind(this)
}
```

事件处理器的代码很简单，它们只是修改了状态，增加或者减少当前的计数：

```
handleDecrement() {
  this.setState({
    counter: this.state.counter - 1,
  })
}

handleIncrement() {
  this.setState({
    counter: this.state.counter + 1,
  })
}
```

最后，在渲染方法中显示当前值，每个按钮的 `onClick` 属性定义好各自的处理器。

```
render() {
  return (
    <div>
      <h1>{this.state.counter}</h1>
      <button onClick={this.handleDecrement}>-</button>
      <button onClick={this.handleIncrement}>+</button>
    </div>
  )
}
```

5.1.1　子组件与父组件的通信（回调函数）

Counter 组件目前没什么大问题，只是有多项职责：

- 将计数器的值保存在状态中；
- 负责显示数据；
- 包含增加和减少计数器值的逻辑。

将组件拆分得更小始终是一个好主意。每个小型组件带有特定逻辑，这样能提升应用的可维护性，并且可以更灵活地应对需求的变更。

思考以下情况：应用的另一部分也需要同样的增减按钮。

复用 Counter 组件中定义的按钮固然很好，但问题在于，如果将按钮移到组件外部，怎样得知何时点击了按钮，从而更新计数器呢？

当子组件需要向父组件推送信息或触发事件时，React 通常采用**回调函数**来实现。

我们来看看其中的工作原理。

先创建 Buttons 组件来显示增减按钮。当点击按钮时，不再触发内部函数，而是触发 props 上传来的函数。

```
const Buttons = ({ onDecrement, onIncrement }) => (
  <div>
    <button onClick={onDecrement}>-</button>
    <button onClick={onIncrement}>+</button>
  </div>
)

Buttons.propTypes = {
  onDecrement: React.PropTypes.func,
  onIncrement: React.PropTypes.func,
}
```

这是一个简单的无状态函数式组件，其内部的 onClick 事件处理器会触发 props 上的函数。

现在看看如何将这个新组件集成到 Counter 组件中。

用新组件替换原有标记即可，并将内部函数传给它，如下所示。

```
render() {
  return (
    <div>
      <h1>{this.state.counter}</h1>
      <Buttons
        onDecrement={this.handleDecrement}
        onIncrement={this.handleIncrement}
      />
    </div>
  )
}
```

其他部分都保持原样，逻辑仍然位于父组件中。

现在按钮组件变得纯粹了，被点击时，它们只能通知自身的拥有者。

因此，每当子组件需要向父组件推送数据或者通知父组件发生了某个事件时，可以传递回调函数，同时将其余逻辑放在父组件中。

5.1.2　公有父组件

现在，Counter 组件的代码看起来好多了，并且 Buttons 组件也可以复用。要让它变得完全整洁，最后一步就是抽离显示逻辑。

为了实现这一点，我们可以创建一个 Display 组件来接收所需的值并在屏幕上显示出来：

```
const Display = ({ counter }) => <h1>{counter}</h1>

Display.propTypes = {
  counter: React.PropTypes.number,
}
```

因为不需要保存任何状态，所以这里再次使用了无状态函数式组件。你可能已经注意到了，其实没必要拆分这个组件，因为它只渲染了一个 h1 元素。然而，你可能会在应用内添加 CSS 类，显示逻辑以根据值改变计数器的颜色，等等。

总的来说，我们的目的是让组件与数据源无关，这样就能在应用各部分的不同数据源下复用组件。

在 Counter 组件中使用新组件很简单，用 Display 组件替换旧标记即可，如下所示：

```
render() {
  return (
    <div>
      <Display counter={this.state.counter} />
      <Buttons
        onDecrement={this.handleDecrement}
```

```
        onIncrement={this.handleIncrement}
      />
    </div>
  )
}
```

如你所见，两个兄弟组件通过公有父组件进行了通信。

当被点击时，`Buttons` 组件会通知父组件，然后父组件会将更新后的值发送给 `Display` 组件。这种模式在 React 中很常见，而且它可以有效地在没有直接联系的组件间共享数据。

数据始终从父组件流向子组件，但子组件可以发送通知给父组件，以便组件树按照新的数据重新渲染。

每次需要让两个没有关联的组件相互通信时，都要找到它们的公有父组件来保存状态。这样一来，当状态更新时，两个子组件都能从 props 接收新数据。

5.2 数据获取

上一节介绍了在树中组件间共享数据的不同模式。

接下来开始学习 React 的数据获取方式，以及这部分逻辑应该放在何处。

本节示例用获取函数发起 Web 请求，后者是 `XMLHttpRequest` 的现代版。

撰写本书时，Chrome 和 Firefox 已经原生实现了获取函数。如果需要支持不同的浏览器，可以使用 GitHub 开发的**获取腻子脚本**（fetch polyfill）。

我们还将使用 GitHub 的公共 API 来加载数据。向以下路径传递用户名会返回 gist 列表：https://api.github.com/users/:username/gists。

gist 就是一些便于开发人员共享的代码片段。

我们要构建的第一个组件是一个简单的 gist 列表，其中包含由用户 gaearon（Dan Abramov）[①] 创建的 gist。

接下来我们探究一些代码并创建一个类。

之所以采用类的方式定义组件，是因为需要保存内部状态并使用生命周期方法。

```
class Gists extends React.Component
```

然后定义一个 `constructor` 方法，在其内部初始化状态：

① Dan Abramov 是 React JS 开发团队的一员，同时也参与开发了 Redux 和 Create React App。——译者注

```
constructor(props) {
  super(props)

  this.state = { gists: [] }
}
```

现在来到最有趣的数据获取部分了。

用于获取数据的代码可以放在两个生命周期钩子中：componentWillMount 和 componentDidMount。

前者会在组件首次渲染前触发，后者则在组件挂载完成后立即触发。

使用前者似乎是正确的做法，毕竟我们希望尽快加载数据，不过需要注意一点。

实际上，服务端渲染和客户端渲染都会触发 componentWillMount 函数。

第 8 章将详细介绍服务端渲染。现在你只需要知道，当在服务端渲染组件时，触发异步 API 会带来预料之外的结果。

因此，我们只能用 componentDidMount 钩子，这样就能确保只在浏览器端调用 API 请求。

这也意味着首次渲染过程中的 gist 列表将是空的，可能要用第 2 章中介绍的技巧来显示一个加载动画，本节不对此展开具体介绍。

正如前文所说，我们将用获取函数请求 GitHub API，以获取 gaearon 的 gist 列表：

```
componentDidMount() {
  fetch('https://api.github.com/users/gaearon/gists')
    .then(response => response.json())
    .then(gists => this.setState({ gists }))
}
```

这段代码的含义是，当 componentDidMount 钩子被触发时，调用获取函数访问 GitHub API。

获取函数返回一个 Promise 对象，当它变为已完成状态时会返回响应对象，而调用该对象的 JSON 方法就能取得响应本身的 JSON 内容。

解析 JSON 内容并返回结果后，就能在组件的内部状态中保存原始的 gist 数据，以便渲染方法使用它们。

```
render() {
  return (
    <ul>
      {this.state.gists.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
}
```

渲染方法很简单，只要遍历 gist 列表并将每项 gist 映射为<li>元素来显示各项的描述即可。

你可能已经注意到了<li>元素的 key 属性。使用它是出于性能方面的考虑，读完本书后你就会理解其中的原因。

如果试图移除 key 属性，React 会在浏览器的控制台内给出警告。

组件可以工作正常，不过正如之前所说，我们可以将渲染逻辑放入一个独立的子组件，以便它更简洁，更便于测试。

移动组件不是什么难题，毕竟我们已经知道了位于应用各个部分的组件如何从其父组件获取所需的数据。

现在，常见的一个需求是多处代码都要从 API 获取数据，但我们并不想复制代码。

要想从组件中移除数据逻辑并在整个应用中复用，其中一个解决方案就是创建高阶组件。

5

在本示例中，高阶组件会代替增强后的组件来加载数据，然后以 prop 的形式向子组件提供数据。

我们来看看具体怎么做。

我们已经知道，高阶组件其实就是函数，它接受组件和一些其他参数，然后返回添加了某些特殊行为的新组件。

接下来我们先通过偏函数写法接受其他参数，然后将实际组件作为第二个参数：

```
const withData = url => Component => (...)
```

withData 函数的命名遵循了 with*前缀模式。

该函数的参数为获取数据的 URL 以及需要数据的组件。

这种做法和前面的示例很像，只不过现在将 URL 作为参数，并且子组件放入了渲染方法。

该函数返回的类如下所示：

```
class extends React.Component
```

它的构造器设置了初始的空状态。

注意，此处用于保存数据的属性名为 data，因为我们要开发通用组件，不希望它只能绑定某种特定格式的对象或集合：

```
constructor(props) {
  super(props)
```

```
    this.state = { data: [] }
  }
```

`componentDidMount` 钩子触发获取函数，将服务端返回的数据转换成 JSON 对象并保存在状态中：

```
componentDidMount() {
  fetch(url)
    .then(response => response.json())
    .then(data => this.setState({ data }))
}
```

需要注意的是，现在使用高阶组件传入的第一个参数来设置 URL。

这样就能复用它向任何 API 路径发起请求了。

最后，为了使高阶组件更加直观，渲染给定组件时展开 props 对象。

同时展开状态，以便向子组件传递 JSON 数据：

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

非常好，高阶组件完工了。

现在我们可以封装应用的任何组件，为它们提供从任何 URL 获取的数据。

我们来看一下具体用法。

首先，创建一个不含数据获取逻辑的组件，它只负责接收数据并像最初示例那样用标记显示出来：

```
const List = ({ data: gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

List.propTypes = {
  data: React.PropTypes.array,
}
```

以上代码段用到了无状态函数式组件，因为我们不需要保存任何状态，也不需要定义事件处理器，这样做往往具有很多好处。

名为 `data` 的 prop 包含了 API 返回的响应，这种通用命名对本组件没什么用处，好在可以通过 ES2015 语法很方便地将它重命名得更有意义。

接着我们来看如何使用高阶组件 `withData`，使其将数据传给刚刚创建的 `List` 组件。

得益于偏函数写法，我们可以先定制高阶组件来发起特定请求，然后再多次复用它，如下所示：

```
const withGists = withData(
  'https://api.github.com/users/gaearon/gists'
)
```

这种做法妙在可以用新的 `withGists` 函数封装任何组件，不用重复指定 URL 就可以接收 gaearon 的 gist 列表。

最后，封装组件并获得新组件：

```
const ListWithGists = withGists(List)
```

现在我们可以将增强后的组件用在应用的任何地方，并且它都能正常运行。

高阶组件 `withData` 很棒，但只能加载静态 URL，而真实的 URL 通常取决于参数或 prop。

遗憾的是，在使用高阶组件时 prop 是未知的，因此在发起 API 请求前，一旦获取 prop 就需要触发某个钩子函数。

可以修改高阶组件，让它接受两种类型的 URL 参数：一种是当前实现的字符串类型，另一种是函数，它接受组件的 prop 并根据传入的参数返回 URL。

实现这一点很简单，修改 `componentDidMount` 钩子即可，如下所示：

```
componentDidMount() {
  const endpoint = typeof url === 'function'
    ? url(this.props)
    : url

  fetch(endpoint)
    .then(response => response.json())
    .then(data => this.setState({ data }))
}
```

如果 URL 是函数，就以 props 为参数来执行它；如果 URL 是字符串，那么就直接使用。

现在可以按照如下方式使用这个高阶组件：

```
const withGists = withData(
  props => `https://api.github.com/users/${props.username}/gists`
)
```

可以在组件的 prop 中设置 `username` 参数，用于加载 gist。

```
<ListWithGists username="gaearon" />
```

5.3 react-refetch

现在，高阶组件可以按预期工作，并且我们可以在代码库中复用它。

问题是，如果需要更多特性，该怎么办？

举例来说，我们可能想向服务端发送一些数据，或者在 prop 改变时重新获取数据。

另外，我们可能不想在 `componentDidMount` 中加载数据，而是采用其他延迟加载策略。

显然，我们可以动手写出需要的一切特性，但有个现成的库已经提供了大量有用的功能。

这个库名叫 `react-refetch`，由 Heroku 的开发人员维护。

我们来看看如何用它有效地替换高阶组件。

上一节中的 `List` 组件是一个无状态函数式组件，它接受 gist 集合并显示各项的描述：

```
const List = ({ data: gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

List.propTypes = {
  data: React.PropTypes.array,
}
```

将该组件封装进高阶组件 `withData` 后，就可以通过 prop 直观地为它提供数据。我们要做的就是增强它，传入 URL 路径即可。

可以用 `react-refetch` 实现同样的目的。首先，安装这个库：

```
npm install react-refetch --save
```

接着导入该模块的 `connect` 函数：

```
import { connect } from 'react-refetch'
```

最后，用 `connect` 这个高阶组件装饰原有组件。

我们利用偏函数技巧将该函数特殊化，然后再复用它：

```
const connectWithGists = connect(({ username }) => ({
  gists: `https://api.github.com/users/${username}/gists`,
}))
```

上述代码的含义如下。

我们将一个函数作为参数传入 `connect` 函数。参数函数接受 props（以及 context）作为参数，这样就能根据组件的当前属性创建动态的 URL。

传入的回调函数返回一个对象，该对象的键名标识相应的请求，键值就是 URL 路径。

URL 并不局限于字符串形式。稍后我们将探讨如何为请求添加多个参数。

现在，可以用刚刚创建的函数来增强组件，如下所示：

```
const ListWithGists = connectWithGists(List)
```

原有组件需要稍作改动才能用于新的高阶组件。

首先，参数名不再是 `data`，而是 `gists`。

实际上，`react-refetch` 库注入的属性与我们在 `connect` 函数中指定的键同名。

`gists` prop 并不是真正的数据，而是一种名为 `PromiseState` 的特殊对象。

5

`PromiseState` 对象是 `Promise` 的同步表示，并且它具备一些很有用的属性，如 `pending` 和 `fulfilled`，可以利用这两个属性来显示加载动画或数据列表。

它还有一个用于处理异常情况的 `rejected` 属性。

当请求变成已满足状态时，可以通过 `value` 属性访问需要加载的数据，并遍历它来显示 gist 列表：

```
const List = ({ gists }) => (
  gists.fulfilled && (
    <ul>
      {gists.value.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
)
```

一旦无状态函数式组件完成渲染，就检查请求是否为已满足状态；如果是，则用 `gists.value` 属性显示列表。

其他一切保持不变。

另外，还需要更新组件的 `propTypes`，并修改所接收 prop 的名称和类型：

```
List.propTypes = {
  gists: React.PropTypes.object,
}
```

既然项目引入了这个库，我们可以为 `List` 组件添加更多功能。

举例来说，可以添加按钮来为 gist 加星。

先从 UI 着手，然后利用 `react-refetch` 库添加真正的 API 请求。

我们不希望在 `List` 组件中增加过多功能，因为其职责就是显示列表。因此，改用子组件渲染每一行。

新组件命名为 `Gist`，我们将它用在循环内部，如下所示：

```
const List = ({ gists }) => (
  gists.fulfilled && (
    <ul>
      {gists.value.map(gist => (
        <Gist key={gist.id} {...gist} />
      ))}
    </ul>
  )
)
```

用 `Gist` 组件替换`<li>`元素即可，接着展开 gist 对象并传给组件，这样它就能接受单个属性，测试和维护也更加方便。

`Gist` 组件是无状态函数式组件，因为加星逻辑交由 `react-refetch` 库处理，所以组件自身不需要任何状态和事件处理器。

组件接受 `description` 属性，并且目前其与原有标记的区别仅在于多了一个+1 按钮。接下来我们将给这个按钮添加一些功能：

```
const Gist = ({ description }) => (
  <li>
    {description}
    <button>+1</button>
  </li>
)

Gist.propTypes = {
  description: React.PropTypes.string,
}
```

为 gist 加星的 URL 路径是 https://api.github.com/gists/:id/star?access_token=:access_token。

此处的`:id` 就是需要加星的 gist 的 ID，`access_token` 是执行加星操作所需的认证令牌。

获取认证令牌的方式有很多，GitHub 的文档提供了很详细的解释。

这些内容同样超出了本书的范畴，因此本节不进行详细介绍。

下一步是在按钮的 `onClick` 事件上添加函数，以便用 gist 的 ID 调用 API。

`react-refetch` 库的 `connect` 函数接受一个函数作为第一个参数。正如我们之前所见，这

个函数需要返回包含请求的对象。

如果这些请求的值是字符串，那么取得 prop 后就会立马开始获取数据。

如果某个请求的值是函数，那么它会传递给组件，并且可以延迟触发。

举例来说，可以在某个事件发生时触发它。

我们来探讨以下代码：

```
const token = 'access_token=123'

const connectWithStar = connect(({ id }) => ({
  star: () => ({
    starResponse: {
      url: `https://api.github.com/gists/${id}/star?${token}`,
      method: 'PUT',
    },
  }),
}))
```


首先，按偏函数方式使用 `connect` 函数，并用 prop `id` 构造 URL。

接着定义请求对象，其中键名是 `star`，键值是函数，并且这个函数也返回请求对象。在本例中，`starResponse` 对应的键值不是简单的字符串，而是包含 `url` 和 `method` 这两个参数的对象。

这样做是因为 `react-refetch` 库默认发出 HTTP `GET` 请求，但本例需要用 `PUT` 请求为 gist 加星。

现在是时候增强组件了：

```
const GistWithStar = connectWithStar(Gist)
```

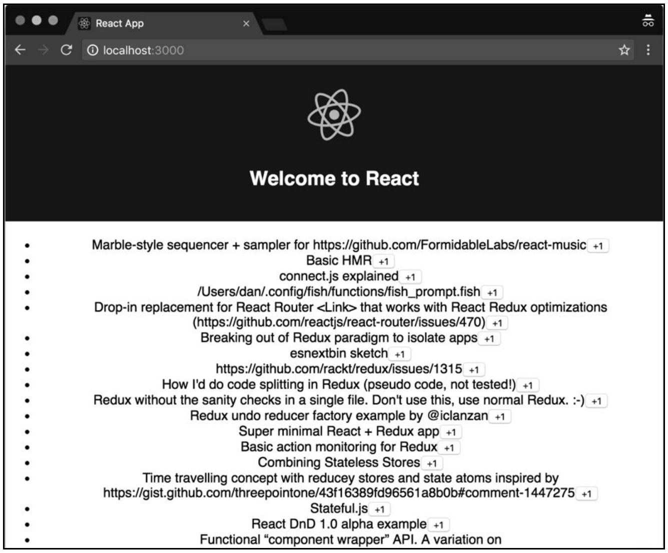
同时在组件内用 `star` 函数发起请求：

```
const Gist = ({ description, star }) => (
  <li>
    {description}
    <button onClick={star}>+1</button>
  </li>
)

Gist.propTypes = {
  description: React.PropTypes.string,
  star: React.PropTypes.func,
}
```

如你所见，代码非常简单；组件接收 `star` 函数，`star` 就是我们在 `connect` 函数中定义的请求名称。点击按钮时触发该函数。

浏览器中的最终结果如下所示：



你应该也注意到了，多亏了 `react-refetch` 库，组件可以保持无状态，而且无须关心它们所触发的操作。

这使得测试更加方便，并且我们可以在不修改子组件的情况下改变高阶组件的实现。

5.4　小结

React 中的数据获取之旅已经结束，现在你已经了解了如何向 API 路径发送数据并从 API 路径接收数据。

我们学习了 React 数据流的工作方式，以及为何它所推行的这种方式能使应用更简洁。

我们探究了几种最常见的模式，它们使子组件可以通过回调函数与父组件通信。

我们还学习了如何利用公有父组件在没有直接关联的组件间共享数据。

5.2 节从加载 GitHub 数据的简单组件起步，利用高阶组件将其改写为可复用组件。

我们掌握了如何从组件中抽象出逻辑，让它们尽量无关，从而提升组件的可测试性。

最后，我们学习了如何利用 `react-refetch` 在组件中应用数据获取模式，并避免重复发明轮子。

下一章将介绍如何在浏览器中有效地使用 React。

第 6 章 为浏览器编写代码

在浏览器中使用 React 时有一些特定用法。举例来说，可以要求用户通过表单输入某些信息，接下来我们将探究 React 如何用各种技巧处理表单。

我们可以实现**自由组件**，以允许表单域保存自己的内部状态。也可以使用受控组件，它们的表单域状态完全由我们控制。

本章还将探究 React 事件的工作原理，以及库如何实现某些高级技巧，以提供跨浏览器的统一接口。接着我们还将讨论 React 团队实现的一些有趣方案，这些方案使得事件系统变得更高效。

介绍完事件后，我们将介绍 ref，了解如何在 React 组件中访问底层 DOM 节点。这项特性很强大，但应该谨慎使用，因为它打破了使得 React 易于使用的某些约定。

介绍完 ref 后，我们将研究如何用 `react-motion` 这类插件以及第三方库便捷地实现 React 动画。最后，我们将了解 SVG 在 React 中的用法是多么简单，以及如何为应用动态地创建可配置的图标。

本章包含如下内容。

- 使用不同的技巧创建 React 表单。
- 监听 DOM 事件，并实现自定义事件处理器。
- 用 ref 对 DOM 节点执行命令式操作的方式。
- 创建跨浏览器运行的简单动画。
- 生成 SVG 的 React 方式。

6.1 表单

用 React 开发真正的应用需要与用户进行交互。如果想要在浏览器中要求用户提供信息，最常见的做法就是使用表单。由于 React 的自身原理及其声明式特性，处理输入控件以及其他表单元素与平时的操作不太一样，不过一旦理解了其中的逻辑，一切就会变得很清晰了。

6.1.1 自由组件

我们从一个最基本的示例开始：显示一张包含输入框和提交按钮的表单。

代码非常简单，如下所示：

```
const Uncontrolled = () => (
  <form>
    <input type="text" />
    <button>Submit</button>
  </form>
)
```

如果在浏览器中执行以上代码，则可以准确得到期望结果：可以在输入框中输入字符，按钮也是可以点击的。本例展示的就是一个自由组件，我们没有在该组件中设置输入框的值，而是让组件自己管理内部状态。

大多数情况下，当提交按钮被点击时，我们想要对输入框元素的值进行操作。

例如，我们可能想要将数据发送到某个 API 路径。

实现这个需求很简单，添加 `onChange` 监听器（本章后面将详细介绍事件监听器）即可。

我们来探究一下添加监听器的意义。

首先，因为需要定义状态和一些函数，所以先将组件从无状态函数改写为类：

```
class Uncontrolled extends React.Component
```

在类的构造器中绑定事件监听器：

```
constructor(props) {
  super(props)

  this.handleChange = this.handleChange.bind(this)
}
```

接着定义事件监听器本身：

```
handleChange({ target }) {
  console.log(target.value)
}
```

事件监听器会接收一个事件对象，其目标属性表示发生事件的元素，这个属性值正是我们想要的。循序渐进地学习很重要，因此一开始先将值打印到控制台，一会儿再保存进状态中。

最后，渲染表单：

```
render() {
  return (
```

```
    <form>
      <input type="text" onChange={this.handleChange} />
      <button>Submit</button>
    </form>
  )
}
```

如果在浏览器中渲染以上组件，并在表单中输入单词 React，我们会在控制台中看到以下内容：

```
R
Re
Rea
Reac
React
```

每当输入框的值改变时，就会触发 handleChange 监听器。因此，每输入一个字符就会调用一次监听器函数。下一步是将用户输入的值保存进状态，以便点击提交按钮时可以获取。

接着修改处理器的实现，移除控制台打印的代码并将值保存进状态，如下所示：

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```


获取表单提交事件与监听输入框改变事件很相似，两者都是某些情况发生时由浏览器触发的事件。

因此，在构造器内增加第二个事件处理器，如下所示：

```
constructor(props) {
  super(props)

  this.state = {
    value: '',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

另外，将状态的值属性的默认值设置为空字符串，以免触发改变事件前就点击了提交按钮。

接着定义 handleSubmit 函数，该函数只会在控制台打印值。在真实场景中，可以用它向 API 路径发送数据，或者将其传给另一个组件。

```
handleSubmit(e) {
  e.preventDefault()

  console.log(this.state.value)
}
```

处理器代码很简单：它只是将当前状态保存的值打印出来。同时我们想要阻止浏览器提交表单时的默认行为，转而执行自定义操作。

这种做法看似很合理，而且单个表单元素的情况下能够完美运行。问题是，多个表单元素的情况下怎么办？假设有数十个表单元素呢？

我们再来看一个手动创建每个表单元素与处理器的基本示例，并讨论如何应用不同层次的优化手段来改进它。

创建一张包含姓和名两个输入框的新表单。这里可以复用刚刚创建的自由类，并修改其构造器，如下所示：

```
constructor(props) {
  super(props)

  this.state = {
    firstName: '',
    lastName: '',
  }

  this.handleChangeFirstName =
    this.handleChangeFirstName.bind(this)
  this.handleChangeLastName = this.handleChangeLastName.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

在状态中初始化这两个输入框的值，并为它们分别定义事件处理器。你可能已经注意到了，这种做法在输入框很多时很难扩展，但重点在于清晰理解问题所在，然后再迁移到更灵活的方案。

现在我们来实现新的处理器，如下所示：

```
handleChangeFirstName({ target }) {
  this.setState({
    firstName: target.value,
  })
}

handleChangeLastName({ target }) {
  this.setState({
    lastName: target.value,
  })
}
```

提交按钮的处理器也要稍作修改，以便在被点击时显示姓与名：

```
handleSubmit(e) {
  e.preventDefault()

  console.log(`${this.state.firstName} ${this.state.lastName}`)
}
```

最后，在 render 方法中编写元素结构：

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input type="text" onChange={this.handleChangeFirstName} />
      <input type="text" onChange={this.handleChangeLastName} />
      <button>Submit</button>
    </form>
  )
}
```

一切准备就绪：在浏览器中运行以上组件将会看到两个输入框。如果在第一个输入框中输入 Dan，第二个中输入 Abramov，那么提交表单后就会在浏览器控制台中看到全名。

这个组件还是完美运行起来了，而且我们可以按这种方式做很多有趣的事，但该组件无法在不编写大量模板代码的情况下应对复杂场景。

接下来我们看看如何对它稍作优化。

我们的基本目的是使用单个改变处理器，这样不用创建新监听器就能添加任意数量的输入框。

回到构造器中并定义单个改变处理器：

```
constructor(props) {
  super(props)

  this.state = {
    firstName: '',
    lastName: '',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

我们可能仍然想要对值进行初始化，本节后面将介绍如何为表单提供预置值。

现在，有趣的地方在于如何修改 onChange 处理器的实现，以便它能处理不同的输入框：

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}
```

正如我们前面所见，事件对象的目标属性表示触发事件的输入框。因此，我们可以将输入框名称及其值作为变量。

接着在渲染方法中为每个输入框设置名称：

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        onChange={this.handleChange}
      />
      <button>Submit</button>
    </form>
  )
}
```

完成。现在可以添加任意数量的输入框，而无须创建额外的处理器。

6.1.2　受控组件

下一步要学习的是如何用某些值预置表单元素，可以从服务端接收这些值，也可以通过 props 从父组件接收。

要想完全理解这个概念，我们还是从一个很简单的无状态函数式组件入手，并逐步改进它。

以下第一个示例展示了输入框内部的预定义值：

```
const Controlled = () => (
  <form>
    <input type="text" value="Hello React" />
    <button>Submit</button>
  </form>
)
```

在浏览器中运行该组件就会发现，它按预期显示了默认值，但不允许我们修改这个值，也不能输入任何字符。

出现这个现象的原因在于，React 允许我们声明想在屏幕上看到的东西，设置固定属性值就会导致始终渲染这个值，不管发生了什么其他操作。但这种行为并不符合真实应用的需求。

如果此时打开控制台，React 本身就会告诉我们用法不对：

```
You provided a `value` prop to a form field without an `onChange`
handler. This will render a read-only field.
(没有`onChange`处理器的情况下，为表单元素提供`value`，prop 将渲染一个只读元素。)
```

事实就是如此。

如果我们希望输入框有默认值，并且输入时可以改变这个值，那么可以使用 defaultValue 属性：

```
const Controlled = () => (
  <form>
    <input type="text" defaultValue="Hello React" />
    <button>Submit</button>
  </form>
)
```

这样输入框在渲染后就会显示 Hello React，而且用户可以输入任何字符来改变其值。这么做可行，也可以很好地运行，但我们想要完全掌控组件值，为此，应该将组件从无状态函数改写为类：

```
class Controlled extends React.Component
```

和往常一样，先定义一个构造器来初始化状态，本例中即为输入框的默认值。同时绑定表单工作所需的事件处理器。

我们只使用单个处理器，它会用名字属性更新状态，我们已经在优化版自由组件的示例中见过这种做法：

```
constructor(props) {
  super(props)

  this.state = {
    firstName: 'Dan',
    lastName: 'Abramov',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

处理器与之前示例中的相同：

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}

handleSubmit(e) {
  e.preventDefault()

  console.log(`${this.state.firstName} ${this.state.lastName}`)
}
```

需要重点修改的是渲染方法。实际上，我们用输入框的值属性来设置初始值，就像一开始做的那样：

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        value={this.state.firstName}
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        value={this.state.lastName}
        onChange={this.handleChange}
      />
      <button>Submit</button>
    </form>
  )
}
```

首次渲染表单时，React 将状态中的初始值作为输入框的值。当用户输入一些内容后，`handleChange` 函数被调用，并将输入框的新值保存进状态。

当状态发生变化时，React 会重新渲染组件，并再次将状态值显示为输入框的当前值。

现在我们完全掌控了表单元素的值，这种模式称为**受控组件**。

6.1.3　JSON schema

现在我们知道了 React 表单的工作原理，为了避免编写大量模板代码并保持代码简洁，接下来我们将学习表单的自动创建。

最流行的解决方案就是由 mozilla-services 所维护的 `react-jsonschema-form`。首先，用 `npm` 安装这个库：

```
npm install --save react-jsonschema-form
```

安装完成后，在组件内导入这个库：

```
import Form from 'react-jsonschema-form'
```

然后定义如下所示的 schema：

```
const schema = {
  type: 'object',
  properties: {
    firstName: { type: 'string', default: 'Dan' },
    lastName: { type: 'string', default: 'Abramov' },
  },
}
```

本书不会涉及 JSON Schema 格式的细节：此处重点在于用配置对象描述表单域，无须创建多个 HTML 元素。

如上例所示，我们将 schema 的类型设为对象，然后声明表单的 `firstName` 和 `lastName` 属性，每个属性都有各自的字符串类型与默认值。

将 schema 对象传递给从库中导入的 `Form` 组件，就会自动生成一张表单。

我们再来看一个简单的无状态函数式组件，并为其迭代添加新特性：

```
const JSONSchemaForm = () => (
  <Form schema={schema} />
)
```

如果在页面中渲染这个组件，那么我们就能看到 schema 中声明的表单元素，以及一个提交按钮。

现在我们希望在提交表单时收到通知，并对表单数据进行某些处理。

首先要做的就是将无状态函数式组件改写为类，然后创建事件处理器：

```
class JSONSchemaForm extends React.Component
```


在构造器中绑定事件处理器：

```
constructor(props) {
  super(props)

  this.handleSubmit = this.handleSubmit.bind(this)
}
```

之前的示例只是将表单数据打印到控制台，而真实应用可能想要将它们发到某个接口路径。

`handleSubmit` 处理器将收到一个对象，该对象的 `formData` 属性包含了表单域的名称和值：

```
handleSubmit({ formData }) {
  console.log(formData)
}
```

最后，渲染方法如下所示：

```
render() {
  return (
    <Form schema={schema} onSubmit={this.handleSubmit} />
  )
}
```

props 中的 schema 就是我们前面定义的 schema 对象。它可以像当前示例一样静态定义，也可以从服务端接收，或者用 props 组合而来。

我们只需要将处理器函数赋值给库组件 `Form` 的 `onSubmit` 回调，就能轻松地创建可用表单。

还有其他回调，比如每当表单元素的值改变时就会触发的 `onChange`，以及提交的表单数据无效时就会触发的 `onError`。

6.2　事件

事件在不同浏览器中的工作方式稍有差别。React 试着抽象了事件的工作方式，以便为开发者提供统一接口。React 的这项特性非常棒，因为我们可以忽略需要兼容哪些浏览器，并编写与浏览器无关的事件处理器和函数。

为了提供这项特性，React 引入了**合成事件**的概念。合成事件对象封装了浏览器提供的原生事件对象，它在任何浏览器中都具有相同属性。

为了给某个节点添加事件监听器，并在触发事件时取得事件对象，我们可以用一个简单的常例来回想一下为 DOM 节点添加事件的方式。实际上，我们用 `on` 加上驼峰式的事件名（如 `onKeyDown`）来定义事件发生时所要触发的回调。命名事件处理器函数的最流行做法就是事件名前加上 `handle` 前缀（如 `handleKeyDown`）。

前面示例中监听表单元素的 `onChange` 事件就应用了这种模式。

我们来重写事件监听器的基础示例，研究如何在同样的组件中更好地组织多个事件。

我们要实现一个简单的按钮，与之前一样，先创建一个类：

```
class Button extends React.Component
```

添加一个构造器来绑定事件监听器：

```
constructor(props) {
  super(props)

  this.handleClick = this.handleClick.bind(this)
}
```

定义事件处理器函数，如下所示：

```
handleClick(syntheticEvent) {
  console.log(syntheticEvent instanceof MouseEvent)
  console.log(syntheticEvent.nativeEvent instanceof MouseEvent)
}
```

如你所见，代码很简单：我们只检查了从 React 接收到的事件对象的类型，以及其封装的原生事件的类型。我们期望前者返回 `false`，后者返回 `true`。

你应该永远不需要访问原生事件对象，不过知道访问方式能在需要时派上用场。最后，在渲

染方法中定义按钮，在其 `onClick` 属性上赋值事件监听器：

```
render() {
  return (
    <button onClick={this.handleClick}>Click me!</button>
  )
}
```

现在，假设我们想要为按钮添加第二个处理器，负责监听双击事件。其中一种做法就是另外创建一个新的处理器，并将它赋值给按钮的 `onDoubleClick` 属性，如下所示：

```
<button
  onClick={this.handleClick}
  onDoubleClick={this.handleDoubleClick}
>
  Click me!
</button>
```

记住，我们始终希望能写更少的模板代码，并避免复制代码。因此，最常见的做法是为每个组件编写**单个事件处理器**，以根据事件的类型触发不同的操作。

Michael Chan 的网站收集了大量的 React 模式，其中就有这个技巧：

http://reactpatterns.com/#event-switch

首先，修改组件的构造器，因为我们现在想要将它绑定到新的通用事件处理器：

```
constructor(props) {
  super(props)

  this.handleEvent = this.handleEvent.bind(this)
}
```

接下来实现这个通用的事件处理器：

```
handleEvent(event) {
  switch (event.type) {
    case 'click':
      console.log('clicked')
      break

    case 'dblclick':
      console.log('double clicked')
      break

    default:
      console.log('unhandled', event.type)
  }
}
```

该处理器接收事件对象，并根据事件类型触发相应的操作。如果每个事件要调用一个函数（比如为了分析）或者某些事件共享相同逻辑，那么这种做法就特别有用了。

最后，将新的事件监听器赋值给 onClick 和 onDoubleClick 属性：

```
render() {
  return (
    <button
      onClick={this.handleEvent}
      onDoubleClick={this.handleEvent}
    >
      Click me!
   </button>
  )
}
```

从现在起，如果需要为同一个组件创建新的事件处理器，无须创建新的方法并绑定，只要在 switch 语句中增加一个事例即可。

React 中的事件还有一些更有趣的东西：合成事件会被回收，并且存在**唯一的全局处理器**。第一个概念是指我们不能保存合成事件稍后再用，因为它在操作完成后就会变成 null。这种做法对性能很有好处，但如果出于某些原因需要在组件状态中保存事件，那么就会产生问题。为了解决这个问题，React 在合成事件中提供了 persist 方法，调用它就能持久保存事件，以便稍后取用。

第二个有趣的实现细节还是与性能相关，并且它涉及 React 为 DOM 添加事件处理器的方式。

只要使用 on 开头的属性，我们就是在向 React 描述期望达成的行为，但库本身不会在底层 DOM 节点上添加真正的事件处理器。

React 实际做的是在根元素上添加单个事件处理器，由于**事件冒泡机制**，这个处理器会监听所有事件。当浏览器触发我们想要的事件时，React 会代表相应组件调用处理器。这个技巧称作**事件代理**，可以优化内存和速度。

6.3 ref

人们喜欢 React 的原因之一就在于它的声明式语法。声明式意味着你只需要描述屏幕在任意给定时刻要显示什么，然后 React 会负责与浏览器通信。这项特性使 React 变得非常易于分析，同时也非常强大。

然而，有时可能仍然需要访问底层 DOM 节点来执行一些命令式操作。应该尽量避免这种做法，大多数情况下都能用符合 React 思想的方案实现相同成果，不过了解这种做法的可能性及其原理也很重要，以便我们拥有更多选择。

假设我们想要创建包含一个输入框和一个按钮的简单表单，并且点击按钮时，输入框获得焦点。

这里要做的就是在浏览器窗口内调用输入框元素的实际 DOM 实例，即输入节点的 `focus` 方法。

创建一个名为 `Focus` 的类，并在构造器中绑定 `handleClick` 方法：

```
class Focus extends React.Component
```

监听按钮的点击事件，以便输入框获得焦点：

```
constructor(props) {
  super(props)

  this.handleClick = this.handleClick.bind(this)
}
```

接着完成 `handleClick` 方法的代码：

```
handleClick() {
  this.element.focus()
}
```

如你所见，这里引用了类的元素属性，并调用了它的 `focus` 方法。

要想理解这个元素属性从何而来，可以查看渲染方法的实现：

```
render() {
  return (
    <form>
      <input
        type="text"
        ref={element => (this.element = element)}
      />
      <button onClick={this.handleClick}>Focus</button>
    </form>
  )
}
```

以上就是逻辑的核心。我们创建了一个表单，其中包含一个输入元素，并在该元素的 ref 属性上定义了一个回调函数。

这个回调函数会在组件挂载时被调用，元素参数表示输入的 DOM 实例。值得注意的是，卸载组件时也会调用这个回调，并传入 `null` 参数来释放内存。回调函数要做的就是保存元素对象的引用，方便以后使用（如在 `handleClick` 方法触发时使用）。接着我们定义了按钮元素及其事件处理器。在浏览器执行以上代码后就会显示带有输入框和按钮的表单，点击按钮后就会按预期那样聚焦输入框。

> 前面提过，一般情况下应该尽量避免使用 ref，因为它们让代码更偏向命令式，可读性与可维护性都变差了。

只能使用 ref 的场景就是在组件中集成其他命令式类库（如 jQuery）时。

值得注意的是，设置非原生组件（以大写字母开头的自定义组件）的 ref 回调时，接收到的回调参数引用不是 DOM 节点实例，而是组件本身的实例。这一点非常强大，因为这允许我们访问子组件的内部实例，不过这样做也很危险，应该尽量避免。

为了展示这种方案的示例，我们来创建两个组件。

- 第一个组件是一个简单的受控输入框，它提供一个 `reset` 函数，用于将输入框的值重置为空字符串。
- 第二个组件是一张表单，其中包含上面的输入框组件，并提供了一个重置按钮，点击后将触发实例方法。

我们先来创建输入框组件：

```
class Input extends React.Component
```

定义一个构造器并设置默认状态（空字符串），绑定 `onChange` 方法以及 `reset` 方法，前者用于控制组件，后者表示组件的公共 API：

```
constructor(props) {
  super(props)

  this.state = {
    value: '',
  }

  this.reset = this.reset.bind(this)
  this.handleChange = this.handleChange.bind(this)
}
```

`reset` 函数很简单，只是将状态设回空字符串：

```
reset() {
  this.setState({
    value: '',
  })
}
```

`handleChange` 方法也很简单，只是保持组件状态与输入元素的当前值同步：

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

最后，在 render 方法中定义 input 元素，以及它的受控值和事件处理器：

```
render() {
  return (
    <input
      type="text"
      value={this.state.value}
      onChange={this.handleChange}
    />
  )
}
```

接着创建 Reset 组件，它会用到上一个组件，并在点击按钮时调用其 reset 方法：

```
class Reset extends React.Component
```

还是照常在构造器内绑定事件处理器：

```
constructor(props) {
  super(props)

  this.handleClick = this.handleClick.bind(this)
}
```

handleClick 方法内的代码很有意思，这里调用了输入框组件实例的 reset 方法：

```
handleClick() {
  this.element.reset()
}
```

最后，按照如下方式定义 render 方法：

```
render() {
  return (
    <form>
      <Input ref={element => (this.element = element)} />
      <button onClick={this.handleClick}>Reset</button>
    </form>
  )
}
```

如你所见，ref 回调在引用节点元素或组件实例上基本一致。

这个特性相当强大，因为我们可以轻易访问组件的方法，不过要小心，这样破坏了封装，给重构带来了困难。假设因为某些原因要重命名 reset 方法，那么你需要检查用到它的所有父组件，并全部修改。

React 非常棒，因为它提供了可以用于所有场景的声明式 API，但同时我们也可以访问底层 DOM 节点和组件实例，以防需要用它们实现更高级的交互和更复杂的结构。

6.4　动画

当谈论 UI 与浏览器时，肯定会涉及动画。

带有动画的 UI 对用户更友好，它们非常重要，能够通知用户某些事情已经或者将要发生。

本节不会详细指导如何创建动画和优美的 UI，实际目标在于提供 React 组件最常用的动画解决方案。

对于 React 这种 UI 库，关键在于可以使开发者很方便地创建并管理动画。React 提供了一个名为 `react-addons-css-transition-group` 的插件，这个组件可以帮助我们声明式地创建动画。再次强调，声明式地执行操作极其强大，这使得代码更易分析，也更方便与团队共享。

我们来看看如何用这个 React 插件给文本添加简单的淡入效果，然后用 `react-motion` 再实现一次相同的效果。`react-motion` 这个第三方库有助于我们更便捷地创建复杂动画。

开始创建动画组件前，先安装插件：

```
npm install --save react-addons-css-transition-group
```

安装完成后，在代码中导入这个组件：

```
import CSSTransitionGroup from 'react-addons-css-transition-group'
```

然后将这个组件封装进想要应用动画的组件：

```
const Transition = () => (
  <CSSTransitionGroup
    transitionName="fade"
    transitionAppear
    transitionAppearTimeout={500}
  >
    <h1>Hello React</h1>
  </CSSTransitionGroup>
)
```

上述代码段中的 props 的相关解释如下。

首先，我们声明了一个 `transitionName` 属性，`ReactCSSTransitionGroup` 会将该属性的值用作子元素的类，这样我们就可以利用 CSS 渐变来创建动画了。

只有一个类不能简单地创建恰到好处的动画，这就是为何要用渐变组合根据动画的状态添加多个类。

在这个示例中，`transitionAppear` 用于告诉组件，子组件出现在屏幕上时开始动画。

因此，库要做的就是当组件完成渲染后，立马向组件添加 `fade-appear` 类（其中 `fade` 就

是 transitionName 的值）。

下一步是添加 fade-appear-active 这个类，这样我们就可以通过 CSS 将动画从初始状态切换到新的状态。

另外还要设置 transitionAppearTimeout 属性来告诉 React 动画的时长，以便 React 在动画完成后才从 DOM 中移除动画元素。

使元素变淡的 CSS 代码如下。

首先，在初始动画状态中定义元素的透明度：

```
.fade-appear {
  opacity: 0.01;
}
```

然后在第二个类中定义渐变，元素添加这个类后就会立刻发生渐变：

```
.fade-appear.fade-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}
```

我们用 ease-in 函数让透明度在 500 毫秒（0.5s）内从 0.01 渐变至 1。

这很简单，我们还能创建更复杂的动画，也可以在组件的不同状态下执行动画。

举例来说，当新元素作为子组件加入渐变组合时，可以应用以-enter 与-enter-active 结尾的类。

移除元素的做法同理。

react-motion

随着动画复杂度的增加、多个动画之间存在关联，或者需要为组件增加更高级的物理行为，此时我们就会意识到渐变组合的帮助有局限，因此需要考虑引入第三方库。

用于创建 React 动画的最流行的库就是由 Cheng Lou 所维护的 react-motion。它非常强大，并且 API 十分简洁易用，能够创建任何类型的动画。

使用之前先安装这个库：

```
npm install --save react-motion
```

安装成功后，导入 Motion 组件和 spring 函数。前者用于封装动画元素，后者用于将一个值从初始状态渐变到最终状态：

```
import { Motion, spring } from 'react-motion'
```

查看以下代码：

```
const Transition = () => (
  <Motion
    defaultStyle={{ opacity: 0.01 }}
    style={{ opacity: spring(1) }}
  >
    {interpolatingStyle => (
      <h1 style={interpolatingStyle}>Hello React</h1>
    )}
  </Motion>
)
```

这段代码有不少有趣之处。

首先，你可能已经注意到了，该组件使用了函数子组件模式（详见第 4 章），这种技巧相当强大，可以定义子组件在运行时接收值。

接着我们可以看到 `Motion` 组件有两个属性：第一个属性 `defaultStyle` 表示初始样式。

我们将透明度设置为 `0.01` 来隐藏元素并开始淡入动画。

样式属性表示最终样式，但这里没有直接设置最终值，而是利用 `spring` 函数，这样样式值就能从初始状态渐变到最终状态。

在 `spring` 函数每次迭代计算的过程中，子函数会即时接收到改变后的样式值，只要将收到的对象赋值给子组件的样式属性，就能看到透明度产生渐变。

这个库还有很多强大的功能，一开始我们学习的是基础概念，这个示例应该讲解得很清楚了。

当你正在做某个项目时，可以比较一下渐变组合和 `react-motion` 这两种方式，以便做出正确选择。

6.5　可扩展矢量图形

最后，**可扩展矢量图形**（scalable vector graphic，SVG）是最有趣的技术之一，可以用于在浏览器中绘制图标和图形。

SVG 非常棒，因为它可以声明式地描述矢量，这刚好与 React 的理念完美匹配。

我们以往经常用图标字体来创建图标，但这项技术的问题众所周知，最大的问题便是它不具备可访问性。再者，用 CSS 定位图标字体相当困难，并且它们无法在所有浏览器中同样美观显示。以上原因就是我们应该在 Web 应用中改用 SVG 的理由。

从 React 的角度来看，从渲染方法输出 `div` 还是 SVG 元素没有什么差别，React 的这一点非常强大。

倾向于选择 SVG 的另一个理由是，可以用 CSS 和 JavaScript 很方便地在运行时修改它们，这使得它们成为了 React 函数式编程的完美候补方案。

因此，如果将组件看作带有 props 参数的函数，就很好理解独立 SVG 图标的创建方式了，我们可以控制传入不同的 props 来实现。

用 React 在 Web 应用中创建 SVG 的常见方式是，将矢量封装进一个 React 组件，并用 props 来定义其动态值。

查看以下绘制蓝色圆圈的简单示例，该示例创建了一个 React 组件并封装了一个 SVG 元素：

```
const Circle = ({ x, y, radius, fill }) => (
  <svg>
    <circle cx={x} cy={y} r={radius} fill={fill} />
  </svg>
)
```

如上所示，可以简单地用无状态函数式组件封装 SVG 标记，组件 props 和 SVG 标记属性一致。

6

SVG 只是模板，传入各种 props 就能在应用中多次复用同一个 `Circle` 组件。

props 的类型定义如下所示：

```
Circle.propTypes = {
  x: React.PropTypes.number,
  y: React.PropTypes.number,
  radius: React.PropTypes.number,
  fill: React.PropTypes.string,
}
```

加上类型定义的好处在于，SVG 及其属性的使用会更加明确，接口变得更加清晰，我们可以准确知道如何配置图标。

示例用法如下所示：

```
<Circle x={20} y={20} radius={20} fill="blue" />
```

显然，我们可以用 React 的全部能力为组件设置一些默认参数，这样一来，即使渲染 `Circle` 图标时没有传入 props，也能显示出内容。

例如，我们可以定义默认颜色：

```
Circle.defaultProps = {
  fill: 'red',
}
```

这种做法对于构建 UI 来说相当强大，尤其是与团队成员分享图标集时，我们希望为图标设置一些默认值，同时允许其他团队决定自己的配置，且不用重新创建相同的 SVG 形状。

但在某些情况下，我们更希望严格保持一致性，确保某些值不可修改。有了 React 后，这个需求就变得非常简单了。

比如，可以将基础的 `circle` 组件封装进 `RedCircle` 组件中，如下所示：

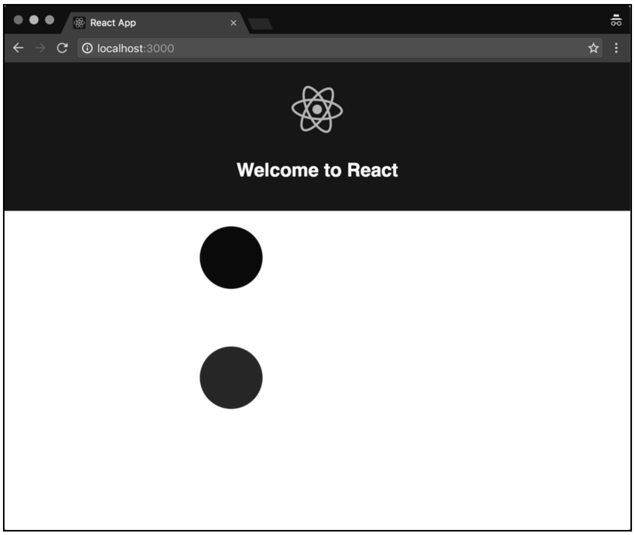
```
const RedCircle = ({ x, y, radius }) => (
  <Circle x={x} y={y} radius={radius} fill="red" />
)
```

这里为颜色设置了默认值并且无法修改，同时，其他 props 直接透传给原来的 `circle` 组件。

props 的类型定义不变，但删除了 `fill` 属性：

```
RedCircle.propTypes = {
  x: React.PropTypes.number,
  y: React.PropTypes.number,
  radius: React.PropTypes.number,
}
```

以下截图展示了 React 用 SVG 生成的蓝色和红色圆圈[①]：



可以用这个技巧创建不同版本的圆圈（如 `SmallCircle` 和 `RightCircle`），并提供构建 UI 所需的一切。

① 在图中显示为深灰色和浅灰色圆圈。读者可以在本书页面（http://www.ituring.com.cn/book/2007）下载书中彩色图片，也可以自行运行代码尝试。——编者注

6.6 小结

本章介绍了针对浏览器编写 React 代码的多种场景，其中包括表单创建、事件处理、动画以及 SVG。

React 提供的声明式编程可以用来管理创建 Web 应用时所要应对的方方面面。

以防需要，React 允许我们访问实际的 DOM 节点，以便对它们执行命令式操作，如果需要在 React 中集成已有的命令式类库，那么这是非常有用的。

下一章将介绍 CSS 与行内样式，并阐明在 JavaScript 中编写 CSS 的意义所在。

第 7 章 美化组件

我们的 React 最佳实践与设计模式之旅已经进行了一大半，现在是时候考虑美化组件了。为了实现这个目标，我们将全面探讨为何常规的 CSS 不是组件样式的最佳方案，然后探讨各种替代方案。

本章将介绍行内样式、Radium 库、CSS 模块以及 Styled Components 库，带你探索 CSS in JavaScript 的奇妙世界。

CSS in JavaScript 的话题十分火热，而且饱受争议，因此，阅读本章时需要大家持有开放的心态。

本章包含如下内容。

- 大型常规 CSS 代码库的常见问题。
- 在 React 中使用行内样式的意义及其缺陷。
- 如何用 Radium 库帮助完善行内样式问题。
- 如何用 Webpack 和 CSS 模块从零搭建一个工程。
- CSS 模块的特性，以及为何它们是解决全局 CSS 的绝佳方案。
- React 组件样式的现代化方案——Styled Component 库。

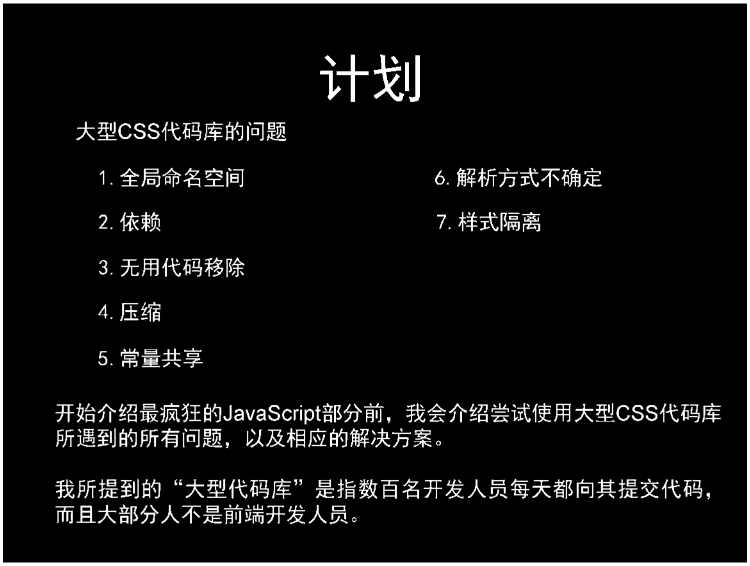
7.1 CSS in JavaScript

2014 年 11 月，当 Christopher Chedeau 在 NationJS 大会上发表演讲“React: CSS in your JS”时，社区中的每个人都认为 React 的组件样式要发生革命性变化了。

以网名 Vjeux 著称的 Christopher 是 Facebook 的员工，参与了 React 的开发。他的演讲中提到了 Facebook 开发人员所遇到的与大型 CSS 代码库相关的一切问题。

这些问题值得好好理解一番，因为有些问题相当常见，也能帮助我们更好地介绍**行内样式**与**局部作用域类名**这些概念。

以下是演讲过程中的一张幻灯片，其中列出了 CSS 的主要问题：



CSS 第一个为人熟知的问题就是所有选择器都是全局的。无论怎样用命名空间或者 BEM 命名法组织样式，最终都会污染全局命名空间，尽管我们都知道这样做不对。这不仅在原则上是不对的，还会引发大型代码库的许多错误，从长远角度来看，可维护性也会变得很糟糕。大型团队的合作有必要知道某个特定的类或者元素是否已经赋过样式，大部分情况下我们更倾向于添加更多类，而不是复用已有的类。

CSS 的第二个问题在于依赖的定义。实际上，很难清晰地声明某个特定组件依赖某段特定的 CSS 代码，并且这段 CSS 要在样式应用前加载完毕。由于样式都是全局的，任何文件中的任何样式都可以应用于任何元素，一不小心就会失控。

前端开发人员往往会用预处理器将 CSS 代码分割成子模块，但最终为浏览器生成的还是一个很大的全局 CSS 文件。由于 CSS 代码库的体积会快速膨胀，失控在所难免。第三个问题是**无用代码移除**。因为很难快速判断哪些样式属于哪个组件，所以删除代码时就非常棘手。实际上，由于 CSS 的层叠特性，删除一个选择器或者规则都能在浏览器中引发意料之外的后果。

使用 CSS 的另一个痛点在于选择器名与类名的压缩，CSS 应用和 JavaScript 应用都要面临这个问题。这看起来似乎很简单，但实际不是，即时应用或客户端拼接的类名更加复杂。

无法压缩和优化类名对性能来说非常糟糕，这对 CSS 文件的大小也有很大影响。

常规的 CSS 也很难做到在样式以及客户端应用间共享常量。举例来说，我们常常需要获取页头的高度，以计算依赖它的其他元素的位置。

我们通常用 JavaScript API 在客户端中阅读各种值，但最理想的做法应该是共享常量，避免在运行时进行繁重的计算。这就是 Vjeux 和 Facebook 其他开发人员试图解决的第五个问题。

第六个问题是 CSS 解析方式的不确定性。实际上，CSS 规则的顺序很重要，如果按需加载 CSS，则无法确保它们的解析顺序，进而导致错误的样式应用于元素。

举例来说，假设我们想要优化请求 CSS 的方式，只在用户浏览到某个特定页面时才加载相关的 CSS。如果与当前页相关的 CSS 中有部分规则也能作用于其他页面的元素，由于它是最后加载的，就会对应用其他部分的样式造成影响。例如，如果返回前一页，用户可能会发现页面 UI 与一开始看到的稍有不同。

要想控制所有样式、规则以及导航路径的各种组合，难度非常大。但话说回来，能够按需加载 CSS 对 Web 应用的性能也有至关重要的意义。

最后，Christopher Chedeau 提出的第七个 CSS 问题与样式隔离有关。实际上，几乎不可能在文件或组件间实现恰当的 CSS 隔离。选择器都是全局的，可以被轻易覆盖。想通过元素上的类名预知其最终样式非常困难，因为样式没有隔离，应用其他部分的规则会影响不相关的元素。

我强烈推荐你观看一遍该演讲，以便了解这个话题的更多细节，虽然它听起来有些强硬，而且存在争议，但仍然会带来很多启发，并促使我们用更开放的心态接触与样式相关的话题。

本次演讲的结论是，为了解决 Facebook 在使用大型 CSS 代码库时遇到的所有问题，可以采用**行内样式**。

下一节将探讨使用行内样式的意义及其优缺点。

7.2　行内样式

React 官方文档推荐开发者在 React 组件上使用行内样式。这听起来似乎很奇怪，因为我们多年来所学的知识都在宣扬关注点分离的重要性，不应该将标记和 CSS 混在一起。

React 试图改变关注点分离这一概念，使其从技术分离向组件分离转变。标记、样式与逻辑耦合得很紧密，应用缺少其中任何一个都无法正常运行，这种情况下将它们独立放入不同文件只是假象上的分离。虽然这确实有助于保持项目结构清晰，但没有带来任何实质的收益。

React 将组件作为应用架构的基础单元，通过组合组件来创建应用。我们可以将组件放到应用的任何位置，并且它们都能渲染出相同的逻辑和 UI。

以上就是将样式放入组件内部的原因之一，而 React 通过行内形式在元素上应用样式的做法也合情合理。

首先我们来看一个示例，它展示了如何利用节点的样式属性为 React 组件应用样式。

我们创建一个带有 Click me! 文本的按钮，并为其添加前景色和背景色：

```
const style = {
  color: 'palevioletred',
  backgroundColor: 'papayawhip',
}

const Button = () => <button style={style}>Click me!</button>
```

如你所见，在 React 中为元素添加行内样式非常简单。只要创建一个对象，它的属性名就是 CSS 规则名，属性值就是常规 CSS 文件中用到的那些值。

唯一的区别在于，为了符合 JavaScript 语法，连字符式的 CSS 规则名必须改为驼峰式，另外属性值必须是字符串，因此需要用引号包裹起来。

厂商前缀方面有一些例外情况。举例来说，如果我们想要定义 webkit 内核的渐变，应该使用 `WebkitTransition` 属性，其中 webkit 前缀以大写字母开头。这项规则对所有厂商前缀有效，但 ms 前缀要以小写字母开头。

数字值也有例外：可以不带引号或度量单位书写它们，默认单位为像素。

以下规则设置高度为 `100` 像素：

```
const style = {
  height: 100,
}
```

行内样式的确可行，而且可以做到常规 CSS 很难实现的需求。举例来说，可以在客户端运行时重新计算某些 CSS 值，你将在接下来的示例中看到这种理念的强大之处。

假设你想要创建一个表单元素，其字体大小随输入值改变。比如，如果输入值为 24，字体大小要变成 24 像素。没有大量精力和重复代码，几乎不可能通过常规 CSS 实现这种效果。

我们来看看行内样式的方案有多简单。

由于需要保存状态以及事件处理器，我们创建一个组件类：

```
class FontSize extends React.Component
```

在构造器中设置状态的默认值并绑定 `handleChange` 处理器，该处理器用于监听输入框的 `onChange` 事件：

```
constructor(props) {
  super(props)

  this.state = {
    value: 16,
```

```
  }

  this.handleChange = this.handleChange.bind(this)
}
```

事件处理器的代码很简单，我们用事件对象的目标属性获取输入框的当前值：

```
handleChange({ target }) {
  this.setState({
    value: Number(target.value),
  })
}
```

最后，渲染类型属性为数字的输入框，这是一个受控组件，因为我们利用状态更新它的值。它还拥有事件处理器，每当输入值改变时就会触发。

我们用输入框的样式属性设置其 `font-size` 样式。如你所见，我们按照 React 的约定书写了驼峰式的 CSS 规则名：

```
render() {
  return (
    <input
      type="number"
      value={this.state.value}
      onChange={this.handleChange}
      style={{ fontSize: this.state.value }}
    />
  )
}
```

渲染以上组件就会看到一个输入框，其字体大小会随着输入值改变。它的工作原理是，当输入值改变时，将新值保存到状态中。修改状态会触发组件重新渲染，然后用新的状态值设置输入框的显示值与字体大小，这项功能简单而又强大。

计算机科学中的每种解决方案都有缺陷，总有需要取舍的地方。遗憾的是，行内样式也有很多问题。

举例来说，行内样式不能使用伪选择器（如`:hover`）和伪元素，这种局限的影响在创建包含交互与动画的 UI 时非常显著。

上述问题也有一些应对方案，比如，可以总是用真实元素代替伪元素，但伪类就必须用 JavaScript 来模拟 CSS 行为，这样做并不理想。

媒体查询同理，它不能在行内样式中使用，而且会使响应式 Web 应用的开发变得很困难。因为样式是用 JavaScript 对象声明的，所以也不能使用样式回退：

```
display: -webkit-flex;
display: flex;
```

实际上，JavaScript 对象不能包含两个同名属性。应该极力避免样式回退，但需要时能用到它再好不过。

CSS 的**动画**特性也无法用行内样式来模拟。对此只能全局定义动画，然后在元素的样式属性中使用。

需要覆盖常规 CSS 的某个样式时，行内样式只能用`!important`关键词来实现，这种做法非常糟糕，因为它会阻止元素添加其他样式。

使用行内样式的最大问题莫过于调试。我们平时习惯在浏览器开发者工具中用类名查找元素，调试并检查哪条样式起作用了。

因为行内样式将所有样式都列在其样式属性中，所以调试检查起来非常麻烦。

举例来说，我们在本节前面创建的按钮的渲染结果如下所示：

```
<button style="color: palevioletred; background-color: papayawhip;">Click
me!</button>
```

就这个按钮而言，阅读代码并不困难，但想象一下拥有数百个元素与数百条样式的情况，你就会发现这个问题相当复杂。

再者，假设你正在调试一个每一项都有相同样式属性的列表。如果实时修改其中一个属性，再查看浏览器中的结果，你会发现只有那一个元素的样式生效，其他兄弟元素保持不变，尽管它们共享相同的样式。

最后很关键的一点是，如果在服务端渲染应用（第 8 章将介绍该话题），使用行内样式会使页面体积变得更大。

行内样式将 CSS 的全部内容都放到标记中，发送给客户端的文件会增加很多，这会降低 Web 应用的呈现速度。

压缩算法可以改善这一点，因为它们可以轻松压缩模式相近的文件，并且在某些情况下，加载关键路径的 CSS 也是不错的方案，但总的来说应该尽量避免这样做。

事实证明，虽然行内样式解决了目标问题，却引发了更多问题。

出于这个原因，社区中出现了不同的工具来试图解决行内样式带来的问题，同时将样式保留在组件中，或者让样式只能作用于局部组件，以获得双赢。

Christopher Chedeau 进行演讲后，许多开发者开始讨论行内样式，也开发了大量解决方案，并进行了很多试验来探索 CSS in JavaScript 的全新方式。

我自己曾尝试过所有方案并创建了一个代码仓库，用每种可用方案分别实现了一个小小的按

钮组件：

https://github.com/MicheleBertoli/css-in-js

一开始只有两三种方案，如今已经超过了 40 种。

本章的后续内容将介绍几种最流行的方案。

7.3　Radium

Radium 是首批试图解决前面提到的行内样式问题的类库之一。它由 Formidable Labs 的一群优秀开发者所维护，并且至今仍是最流行的方案之一。

我们将在本节中探讨 Radium 的工作原理、它解决的问题，以及为何它是 React 设置组件样式的绝佳搭配。

接下来我们将创建一个非常简单的按钮，与本章前面示例中的按钮类似。

我们从无样式的基础按钮起步，接着为它添加一些基础样式、伪类以及媒体查询，并以此来学习 Radium 库的主要特性。

最初的按钮代码如下所示：

```
const Button = () => <button>Click me!</button>
```

首先，用 `npm` 安装 Radium：

```
npm install --save radium
```

安装完成后，导入该库并用它封装按钮：

```
import radium from 'radium'

const Button = () => <button>Click me!</button>

export default radium(Button)
```

`radium` 函数是一个**高阶组件**（详见第 4 章），它可以扩展 `Button` 组件的功能，并返回新的增强组件。

此时在浏览器中渲染按钮不会看到什么特别的东西，因为我们还没有添加任何样式。

现在我们来创建一个很简单的样式对象，用它设置背景色、内边距、大小以及其他一些 CSS 属性。

上一节中说过，要用 JavaScript 对象与驼峰式 CSS 属性来设置 React 的行内样式：

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
}
```

上述代码与普通的 React 行内样式没什么差别，如果按照以下方式将它传递给按钮组件，那么就可以在浏览器中看到按钮上的所有样式都生效了：

```
const Button = () => <button style={styles}>Click me!</button>
```

结果为以下标记：

```
<button data-radium="true" style="background-color: rgb(255, 0, 0); width:
320px; padding: 20px; border-radius: 5px; border: none; outline:
none;">Click me!</button>
```

此处的唯一区别在于元素上多了 `data-radium` 属性，其值为 `true`。

我们知道无法在行内样式中定义伪类。接着我们来看看 Radium 如何解决这个问题。

在 Radium 中使用`:hover` 这样的伪类很简单。

只要在样式对象中创建`:hover` 属性，Radium 就会完成剩余工作：

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
  ':hover': {
    color: '#fff',
  },
}
```

将以上样式对象应用于按钮组件并渲染到屏幕，鼠标移到按钮上就会看到文本从默认的黑色变成白色。

这太棒了：可以一同使用伪类和行内样式了。

然而，如果打开开发者工具，在 Styles 面板选中元素的`:hover` 状态，你会发现什么也没有发生。

能看到悬停特效却无法用 CSS 模拟出来，是因为 Radium 用 JavaScript 来应用和移除样式对象中定义的悬停状态。

如果打开开发者工具，将鼠标悬停在元素上，你会看到样式属性的字符串发生变化，其中动态加入了颜色规则：

```
<button data-radium="true" style="background-color: rgb(255, 0, 0); width:
320px; padding: 20px; border-radius: 5px; border: none; outline: none;
color: rgb(255, 255, 255);">Click me!</button>
```

Radium 的工作原理就是为触发伪类行为的每个事件添加事件处理器，并监听这些事件。

一旦这些事件被触发，Radium 就会改变组件状态，然后组件就根据状态中的正确样式重新渲染。这种做法一开始可能令人感到奇怪，不过没有什么实质缺陷，而且性能方面也没有很明显的差别。

可以再添加新的伪类，如:active，它一样能正常运行：

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
  ':hover': {
    color: '#fff',
  },
  ':active': {
    position: 'relative',
    top: 2,
  },
}
```

Radium 提供的另一个重要特性是媒体查询，它是开发响应式应用的关键所在。当然，Radium 还是通过 JavaScript 为应用带来了这项 CSS 特性。

我们来看看其中的原理。API 非常相似，我们只需要在样式对象上添加新的属性，并在它内部嵌套符合媒体查询规则时必须生效的那些样式：

```
const styles = {
  backgroundColor: '#ff0000',
  width: 320,
  padding: 20,
  borderRadius: 5,
  border: 'none',
  outline: 'none',
  ':hover': {
    color: '#fff',
  },
  ':active': {
    position: 'relative',
    top: 2,
```

```
  },
  '@media (max-width: 480px)': {
    width: 160,
  },
}
```

要想媒体查询生效，还有一个必要步骤：将应用封装进 Radium 提供的 `StyleRoot` 组件。

为了媒体查询可以正常工作，尤其是服务端渲染的情况下，Radium 会将与媒体查询相关的规则注入 DOM 中的一个样式元素，并且所有属性都会添加`!important` 关键词。

这样做是为了在 Radium 计算出匹配的查询条件前，避免页面文档切换不同样式时发生闪烁。将样式放入样式元素，让浏览器照常完成工作，可以避免样式闪烁现象。

因此，需要导入 `StyleRoot` 组件：

```
import { StyleRoot } from 'radium'
```

然后用它封装整个应用：

```
class App extends Component {
  render() {
    return (
      <StyleRoot>
        ...
      </StyleRoot>
    )
  }
}
```


完成以上步骤后，打开开发者工具就能看到 Radium 在 DOM 中注入了以下样式：

```
<style>@media (max-width: 480px){ .rmq-1d8d7428{width: 160px
!important;}}</style>
```

`rmq-1d8d7428` 类已经自动添加到按钮元素上：

```
<button class="rmq-1d8d7428" data-radium="true" style="background-color:
rgb(255, 0, 0); width: 320px; padding: 20px; border-radius: 5px; border:
none; outline: none;">Click me!</button>
```

如果现在调整浏览器窗口大小，可以看到按钮在窄屏下变得更小了，这正是我们所预想的效果。

7.4　CSS 模块

如果你认为行内样式方案不适合自己的项目与团队，但仍然希望尽量紧密结合样式与组件，那么还有一个名为 CSS 模块的方案供你选择。

7.4.1　Webpack

在探索 CSS 模块并学习其工作原理前，先了解一下它的创建过程及支持它的工具是很有必要的。

我们在第 2 章中探讨了如何编写 ES2015 代码，并用 Babel 及其预设配置进行转译。随着应用体积的膨胀，你可能还想要将代码库拆分成模块。

可以用 Browserify 或 Webpack 这类工具将应用拆分成小型模块，以便按需导入，同时仍然为浏览器生成一个大文件。这类工具称作**模块打包器**，它们的工作就是将应用的所有依赖加载到单个打包文件中，以便于在浏览器中运行，因为浏览器（尚且）没有模块的概念。

Webpack 在 React 领域特别流行，因为它提供了很多有趣且好用的特性，第一个就是**加载器**的概念。Webpack 理论上可以加载除 JavaScript 以外的任何依赖，只要有对应的加载器。举例来说，可以在打包文件中加载 JSON 文件、图片以及其他资源。

2015 年 5 月，CSS 模块的创建者之一 Mark Dalgleish 发现 Webpack 还能打包导入 CSS，于是他推动了这一概念的发展。

他认为，既然 CSS 能以局部形式导入组件，那么导入的所有类名也可以带上局部作用域。他的“The End of Global CSS”一文详细解释了这个概念。

7.4.2　搭建项目

本节将介绍如何搭建一个简单的 Webpack 应用，其中会涉及用 Babel 转译 JavaScript，以及用 CSS 模块加载打包局部作用域的 CSS。我们还将学习 CSS 模块的全部特性，并探究它能解决的问题。首先，进入一个空文件夹，并执行以下命令：

```
npm init
```

这条命令会创建包含一些默认配置的 package.json 文件。

现在开始安装依赖，先是 Webpack，其次是 `webpack-dev-server`，我们用它们在本地运行应用并实时生成打包文件：

```
npm install --save-dev webpack webpack-dev-server
```

安装好 Webpack 后，就可以安装 Babel 及其加载器了。因为是用 Webpack 生成打包文件的，所以我们还要在 Webpack 的内部用 Babel 加载器来转译 ES2015 代码：

```
npm install --save-dev babel-loader babel-core babel-preset-es2015
babel-preset-react
```

最后，安装 `style-loader` 和 `css-loader`，这两个加载器用于启用 CSS 模块：

```
npm install --save-dev style-loader CSS-loader
```

为了让一切更方便，还需要安装 `html-webpack-plugin` 插件来创建 HTML 页面，以便实时托管 JavaScript 应用。有了它，我们就不再需要创建常规的 HTML 文件，只编辑 Webpack 配置即可。

```
npm install --save-dev html-webpack-plugin
```

最后但也很重要的一步是，安装 `react` 和 `react-dom`，以用于接下来的简单示例：

```
npm install --save react react-dom
```

现在所有依赖都安装完毕了，可以开始配置来运行应用了。

首先，在 package.json 中添加一条 `npm` 脚本，以启动 `webpack-dev-server` 来托管开发环境中的应用：

```
"scripts": {
  "start": "webpack-dev-server"
},
```

Webpack 需要从配置文件中得知如何处理应用中不同类型的依赖。为此，我们创建了一个 webpack.config.js 文件，该文件会导出一个对象：

```
module.exports = { }
```

导出的对象就是 Webpack 用来生成打包文件的配置对象，根据项目大小和特点，这个对象可以具有不同的属性。

因为我们希望示例尽可能简单，所以只添加三个属性。

第一个属性 `entry` 告诉 Webpack 哪个是应用的主文件：

```
entry: './index.js',
```

第二个属性 `module` 告诉 Webpack 如何加载外部依赖。它有一个名为 `loaders` 的属性，用于为每种文件类型指定加载器：

```
module: {
  loaders: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      loader: 'babel',
      query: {
        presets: ['es2015', 'react'],
      }
    },
    {
      test: /\.css$/,
```

```
        loader: 'style!css?modules',
      },
    ],
  },
```

以上配置表示用 `babel-loader` 加载符合正则表达式.js 的文件，这样它们就能转译并载入打包文件。

你可能还注意到其中也包含预设配置。我们在第 2 章中学习过，预设配置就是配置选项的集合，它们指导 Babel 处理不同语法类型（比如 JSX）。

`loaders` 数组的第二项告诉 Webpack 如何处理导入的 CSS 文件，它用到了 `css-loader`，同时开启 `modules` 标记来启用 CSS 模块。转换结果会传给 `style-loader`，后者负责将样式注入页面头部。

最后，启用 HTML 插件来生成页面，用之前指定的入口文件路径自动添加脚本标签：

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
...
plugins: [new HtmlWebpackPlugin()]
```

现在一切就绪，在终端中执行 `npm start` 命令，并在浏览器中访问 http://localhost:8080，然后就可以看到以下标记被成功托管：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Webpack App</title>
  </head>
  <body>
  <script type="text/javascript" src="bundle.js"></script></body>
</html>
```

7.4.3　局部作用域的 CSS

现在可以开始开发应用了，它包括一个和之前示例中一样的简单按钮。我们会用它展示 CSS 模块的全部特性。

创建 index.js 文件，将其作为 Webpack 配置中指定的入口文件，同时导入 `React` 和 `ReactDOM`：

```
import React from 'react'
import ReactDOM from 'react-dom'
```

接着创建按钮。和往常一样，按钮一开始没有任何样式，之后再逐步加上样式：

```
const Button = () => <button>Click me!</button>
```

最后，将按钮渲染到 DOM 中：

```
ReactDOM.render(<Button />, document.body)
```

注意，直接在主体元素下渲染 React 组件的做法不好，本示例只是出于简洁的考虑才这样做。

现在，我们要为按钮添加一些样式：背景色、大小等。

创建一个名为 index.css 的常规 CSS 文件，并编写以下类：

```
.button {
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

我们曾提过可以用 CSS 模块向 JavaScript 中导入 CSS 文件；接下来我们来看看具体用法。

在定义按钮组件的 index.js 中添加以下代码：

```
import styles from './index.css'
```

`import` 语句会导入一个样式对象，其所有属性就是 index.css 中定义的类。

运行 `console.log`（样式），开发者工具中会输出以下对象：

```
{
  button: "_2wpxM3yizfwbWee6k0UlD4"
}
```

这样看来，我们得到的对象的属性名就是 CSS 类名，而属性值（很显然）是随机字符串。后面我们会发现它们并不是随机的，现在先来看看这个对象有什么作用。

可以用这个对象设置按钮的 `className` 属性，如下所示：

```
const Button = () => (
  <button className={styles.button}>Click me!</button>
)
```

回到浏览器就能看到 index.css 所定义的样式在按钮上生效了。

这没什么神奇的，因为只要检查开发者工具就会发现，元素上新增的类名就是代码导入的样式对象的属性字符串：

```
<button class="_2wpxM3yizfwbWee6k0UlD4">Click me!</button>
```

如果查看页面头部，我们还会发现相同的类名已经注入页面：

```
<style type="text/css">
._2wpxM3yizfwbWee6k0UlD4 {
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
</style>
```

以上就是 CSS 和样式这两个加载器所做的工作。

css-loader 允许你在 JavaScript 模块中导入 CSS 文件，并且启用 modules 标记时，所有类名都只作用于导入它们的模块。

前面提到过，导入的字符串并不是随机的，而是根据文件散列值和其他一些参数生成的，它在代码库中是唯一的。

最后，style-loader 接收 CSS 模块转换的结果，并将样式注入页面头部。

这种用法非常强大，因为我们拥有了 CSS 的完整能力及表现性，又结合了局部作用域类名与显式依赖的优点。

本章开头提过，CSS 是全局的，这一点使其在大型应用中难以维护。有了 CSS 模块，类名就有局部作用域了，不会与应用其他部分的类名发生冲突，因此可以确保页面效果的确定性。

此外，在组件内显式导入 CSS 依赖能够帮助我们搞清楚组件和 CSS 的关系。这在移除无用代码方面也很有用，因为删除某个组件时能够准确找到其所用的 CSS。

CSS 模块就是常规的 CSS，因此可以使用伪类、媒体查询和动画。

举例来说，可以添加以下的 CSS 规则：

```
.button:hover {
  color: #fff;
}

.button:active {
  position: relative;
  top: 2px;
}

@media (max-width: 480px) {
  .button {
    width: 160px
  }
}
```

上述代码会转换为如下代码并注入页面文档：

```
._2wpxM3yizfwbWee6k0UlD4:hover {
  color: #fff;
}

._2wpxM3yizfwbWee6k0UlD4:active {
  position: relative;
  top: 2px;
}

@media (max-width: 480px) {
  ._2wpxM3yizfwbWee6k0UlD4 {
    width: 160px
  }
}
```

生成的类名在按钮组件使用的每个地方都不一样，这就确保它们能像预期一样可靠，并且只作用于局部。

你可能已经注意到了，这些类名的作用很棒，但调试起来相当困难，因为我们无法轻易辨别出散列值是哪个类名生成的。

当处于开发模式时，可以添加一个特殊的配置参数，通过它就可以选择作用域类名的生成模式。

举例来说，我们可以改变加载器的值，如下所示：

```
loader: 'style!css?modules&localIdentName=[local]--[hash:base64:5]',
```

`localIdentName` 就是上面所说的参数，`[local]`与`[hash:base64:5]`分别是原有类名和 5 个字符的散列值的占位符。

还有其他占位符，`[path]`表示 CSS 文件的路径，`[name]`表示 CSS 源文件的名称。

启用以上配置后，浏览器中的结果如下所示：

```
<button class="button--2wpxM">Click me!</button>
```

这样既提高了可读性，也更方便调试。

生产环境下不需要这样的类名，更注重性能，因此我们想要更简短的类名和散列值。

用 Webpack 实现这种需求非常简单，因为可以有多个配置文件，以便用于应用生命周期的不同阶段。另外在生产环境下，不要直接将打包文件中的 CSS 注入浏览器，而要将它们提取出来，这样我们就能得到更小的文件包，并将 CSS 缓存到 CDN，从而获得更好的性能。

为此，我们需要安装另一个名为 `extract-text-plugin` 的 Webpack 插件，它可以将 CSS 模块生成的所有作用域类放入一个真正的 CSS 文件。

CSS 模块的其他一些特性也很值得一提。

第一个就是 global 关键词。给任何类添加:global 前缀，意味着请求 CSS 模块不要为当前选择器加上局部作用域。

举例来说，修改 CSS 代码，如下所示：

```
:global .button {
  ...
}
```

输出结果如下所示：

```
.button {
  ...
}
```

这样做的好处在于，你可以应用不需要局部作用域的样式，比如第三方组件。

我最喜欢的 CSS 模块特性是**组合**。有了它，就可以从同个文件或者外部依赖中引用类名，将其他类的所有样式应用于一个元素。

例如，可以从 button 类的样式规则中提取设置背景色为红色的规则，然后放入独立的类，如下所示：

```
.background-red {
  background-color: #ff0000;
}
```

接着就能按以下方式将它和 button 类组合起来：

```
.button {
  composes: background-red;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

最终，button 类的所有规则以及 composes 声明的所有规则都能作用于元素。

这个特性非常强大，而且原理很巧妙。你可能以为它和 SASS 的@extend 方法一样，只是将组合类复制到引用它们的位置，其实不是这样。简单来讲，所有组合类名都是逐个应用到 DOM 中的组件上。

拿我们的示例来说，代码如下所示：

```
<button class="_2wpxM3yizfwbWee6k0UlD4 Sf8w9cFdQXdRV_i9dgcOq">Click
me!</button>
```

注入页面的 CSS 如下所示：

```
.Sf8w9cFdQXdRV_i9dgcOq {
  background-color: #ff0000;
}

._2wpxM3yizfwbWee6k0UlD4 {
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
}
```

7.4.4　原子级 CSS 模块

现在我们很清楚组合的原理以及为何它是 CSS 模块的一项强大特性。当开始撰写这本书时，我们曾在我就职的 YPlan 公司中试着结合 `composes` 的功能和**原子级 CSS**（又以**函数式 CSS** 著称）的灵活性。

原子级 CSS 是 CSS 的一种使用方式，即每个类只有一条规则。

例如，可以创建一个类来设置底部外边距为 0：

```
.mb0 {
  margin-bottom: 0;
}
```

可以用另一个类设置 `font-weight` 属性为 600：

```
.fw6 {
  font-weight: 600;
}
```

然后将这些原子类用在元素上：

```
<h2 class="mb0 fw6">Hello React</h2>
```

这种技巧存在争议，但很高效。要决定使用它并不容易，因为它最终会导致标记上有太多类，进而导致很难预测最终结果。你可能想到了，这和行内样式很相似，因为一个类只有一条规则，只不过规则名换成了短一些的类名而已。

原子级 CSS 的最大争议在于将 CSS 的样式逻辑移到了标记中，这样是错误的。类是在 CSS 文件中定义的，却在视图层组合，每次修改元素的样式都要同时修改标记。

另一方面，我们试着稍微使用了原子级 CSS，并发现它可以超快地搭建原型。

其实，只要所有基本规则都定好，将这些类应用于元素或者用它们生成新的样式都非常快，

这是一大优点。其次，使用原子级 CSS 可以控制 CSS 文件的大小，因为创建新组件时可以复用已有类的样式，不需要编写新样式，这对性能很有好处。

因此，我们尝试用 CSS 模块来解决原子级 CSS 的问题，并将这种技巧称作**原子级 CSS 模块**。

从本质上来说，以创建基础 CSS 类（如 `mb0`）开始，接着用 CSS 模块将它们组合成占位类，而不是将它们逐个用到标记上。

查看以下示例：

```
.title {
  composes: mb0 fw6;
}
```

以及：

```
<h2 className={styles.title}>Hello React</h2>
```

这种做法非常好，因为样式逻辑仍然保留在 CSS 中，同时 CSS 模块 `composes` 帮助将所有单个类应用到标记上。

上述代码的渲染结果如下所示：

```
<h2 class="title--3JCJR mb0--21SyP fw6--1JRhZ">Hello React</h2>
```

此处的 `title`、`mb0` 以及 `fw6` 都是自动加到元素上的。并且它们都只作用于局部，因此我们就用上了 CSS 模块的所有优势。

7.4.5　React CSS 模块

最后，还有一个很重要的库可以帮助我们更好地使用 CSS 模块。你可能已经注意到我们一直用样式对象加载 CSS 的所有类，这是因为 JavaScript 不支持连字符属性，所以类名只能是驼峰式的。

另外，引用 CSS 文件中不存在的类名不会有任何提示，但类名列表会多出一个 `undefined` 属性。

为了各种有用的特性，我们希望引入一个第三方包，以便 CSS 模块用起来更顺畅。

现在我们来看一下具体用法，回到本节前面用到纯 CSS 模块的 index.js 文件中，将它换成使用 React CSS 模块。

第三方包名为 `react-css-modules`，首先要安装它：

```
npm install --save react-css-modules
```

安装完成后，在 index.js 中导入它：

```
import cssModules from 'react-css-modules'
```

可以将它作为高阶组件，传入想要得到增强的 `Button` 组件以及从 CSS 导入的样式对象：

```
const EnhancedButton = cssModules(Button, styles)
```

然后要改变按钮组件的实现，不再使用样式对象。有了 React CSS 模块，就可以使用 `styleName` 属性，它会转换为常规的类。

这里的妙处在于可以使用字符串形式的类名（如`"button"`）：

```
const Button = () => <button styleName="button">Click me!</button>
```

现在将 `EnhancedButton` 组件渲染到 DOM 中，我们会看到效果和之前完全一样，这就表示这个库起作用了。

如果试着将 `styleName` 属性指向不存在的类名，如下所示：

```
const Button = () => (
  <button styleName="button1">Click me!</button>
)
```

浏览器的控制台会抛出以下错误：

```
Uncaught Error: "button1" CSS module is undefined.
```

当代码库增大并且多名开发人员同时开发不同组件和样式时，这一点特别有用。

7.5 Styled Component

这个库的前景非常广阔，它考虑到了其他组件样式库遇到的所有问题。

CSS in JavaScript 的开发模式已经有很多不同的实现方式，我们也尝试了许多解决方案，现在需要取各家之长，并在它们的基础上总结出更好的方案。

JavaScript 社区的著名开发者 Glenn Maddern 和 Max Stoiberg 设计并开发了 Styled Component 库。

它用现代手段解决组件样式问题，并在 React 中运用了 ES2015 的最新特性和其他高级技巧，实现了完善的样式方案。

接下来我们要用 Styled Component 创建和之前一样的按钮，并检查我们想要的各种 CSS 特性能否使用（如伪类和媒体查询）。

首先，执行以下命令来安装它：

```
npm install --save styled-components
```

安装完成后，将它导入组件文件中：

```
import styled from 'styled-components'
```

这样就可以用样式化函数创建任何元素，形式如 `styled.elementName`，其中 `elementName` 可以是 `div`、按钮或者任何有效的其他 DOM 元素。

接着为将要创建的元素定义样式。为此，我们需要用到 ES2015 的**标签模板字面量**特性，它可以向函数传递未经插值的模板字符串。

这意味着函数可以接收包括所有 JavaScript 表达式的真正模板，这使得该库可以用 JavaScript 的全部能力为元素添加样式。

我们来创建带有基础样式的简单按钮：

```
const Button = styled.button
  backgroundColor: #ff0000;
  width: 320px;
  padding: 20px;
  borderRadius: 5px;
  border: none;
  outline: none;
`
```

这种看似奇怪的语法会返回普通的 React 组件 `Button`，它渲染了一个按钮元素，并加上了模板中定义的样式。先创建唯一的类名，再将它加到元素上，最后向页面文档头部注入相应的样式。至此，样式生效了。

渲染的组件如下所示：

```
<button class="kYvFOg">Click me!</button>
```

页面上添加的样式如下所示：

```
.kYvFOg {
    background-color: #ff0000;
    width: 320px;
    padding: 20px;
    border-radius: 5px;
    border: none;
    outline: none;
}
```

Styled Component 库的优点在于支持几乎所有的 CSS 特性，因此它对真实应用来说是一个很好的备选方案。

比如，它支持 SASS 风格的伪类语法：

```
const Button = styled.button`
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
  &:hover {
    color: #fff;
  }
  &:active {
    position: relative;
    top: 2px;
  }
```

它也支持媒体查询：

```
const Button = styled.button`
  background-color: #ff0000;
  width: 320px;
  padding: 20px;
  border-radius: 5px;
  border: none;
  outline: none;
  &:hover {
    color: #fff;
  }
  &:active {
    position: relative;
    top: 2px;
  }
  @media (max-width: 480px) {
    width: 160px;
  }
`
```

我们的项目还能用到这个库的很多其他特性。

举例来说，创建按钮组件后，可以很方便地覆盖其样式，并设置不同属性来多次复用该组件。

还可以在模板内根据组件所接收的 props 相应地改变样式。

这个库的另一项绝佳特性是**主题**。将组件封装在 ThemeProvider 组件中，可以为组件树注入主题属性，当要和其他组件共享一部分样式，剩下部分取决于当前选中主题时，创建 UI 会变得非常方便。

7.6 小结

本章探讨了许多有趣的话题。我们一开始介绍了大型 CSS 代码库存在的问题，尤其是 Facebook 开发人员编写 CSS 时所遇见的那些问题。

我们学习了 React 行内样式的工作原理，以及将样式放入组件的原因。同时我们也探究了行内样式的局限性。

接着我们提到了 Radium，这个库解决了行内样式的主要问题，使我们以 CSS in JavaScript 模式编写样式时能够定义清晰的接口。有些人认为行内样式方案很糟糕，为此我们又探讨了 CSS 模块的领域，并从零搭建了一个简单项目。

将 CSS 文件导入组件使得依赖关系更加清晰，另外局部作用域的类名也避免了样式冲突。我们还见识了 CSS 模块 `composes` 的强大之处，并学习了如何用它和原子级 CSS 搭配创建快速原型框架。

最后，我们稍微了解了一下 Styled Component 库，这个前景广阔的库将完全改变组件样式的编写方式。

第 8 章

服务端渲染的乐趣与益处

构建 React 应用的下一步是学习服务端渲染的原理及益处。**通用应用**更有利于搜索引擎优化（search engine optimization，SEO），而且能促使前后端共享知识。

它们还能显著提升 Web 应用的感知速度，这往往有助于提升用户转化率。然而，为 React 应用启用服务端渲染也是有代价的，应该仔细考虑是否真的需要进行。

本章将介绍如何搭建服务端渲染应用。阅读完相关内容后，你将掌握通用应用的构建方法，并理解这项技术的优缺点。

本章包含如下内容。

- 通用应用是什么。
- 启用服务端渲染的理由。
- 用 React 创建简单的静态服务端渲染应用。
- 了解服务端渲染的数据获取方式，并理解**脱离**和**注回**等概念。
- 用 Zeit 开发的 Next.js 可以轻松创建能同时在服务端和客户端运行的 React 应用。

8.1　通用应用

提到 JavaScript Web 应用，我们往往会想到在浏览器中运行的客户端代码。

这类应用往往通过服务端返回带有`<script>`标签的空 HTML 页面进行加载。加载完毕后，应用会操作浏览器中的 DOM 来显示 UI，并与用户交互。这种模式在近几年比较流行，而且以后大多数应用仍然会采用这样的方式。

到目前为止，本书介绍了 React 组件为应用开发带来的便利，以及它们在浏览器中的工作原理。我们还没见识到 React 如何在服务端渲染相同的组件，这个强大的特性称作**服务端渲染**（server-side rendering，SSR）。

在开始介绍细节前，我们先试着理解开发同时在服务端和客户端渲染的应用是什么意思。多年以来，我们一直都是为服务端和客户端分别开发不同的应用，比如，服务端用 Django 应用渲染视图，客户端则运行 Backbone 或 jQuery 这类 JavaScript 框架。这些相互独立的应用往往需要两支掌握不同技术栈的开发团队来维护。如果服务端渲染完成的页面需要和客户端应用共享数据，只能在`<script>`标签中注入变量。不同的语言和平台导致应用无法在两端间共享模型或视图这类通用信息。

自 2009 年 Node.js 发布后，得益于 Express 这类 Web 应用框架的出现，JavaScript 在服务端获得了大量关注和普及。

两端使用同一种语言不但有利于开发人员复用已有的知识，而且服务端与客户端之间也有了更多共享代码的途径。

以 React 为例，同构 Web 应用这一概念在 JavaScript 社区内越来越流行。

同构应用就是指应用在服务端和客户端看起来一模一样。

实际上，同一种语言开发两种应用意味着大部分逻辑都能共享，这带来了很多机会。代码库分析将会更简单，同时也避免了不必要的代码重复。

React 促使这个概念更进一步，它提供的 API 可以非常简单地在服务端渲染组件，并直观应用一切所需的逻辑使页面在浏览器中可交互（如事件处理器）。

同构一词并不符合这种场景，因为就 React 而言，（在两端运行的）应用是完全一样的，因此 React Router 的创造者之一 Michael Jackson 提出了更符合这种模式的命名：**通用**。

通用应用是指应用的代码可以同时用于服务端和客户端。

本章将介绍为何应该考虑开发通用应用，以及如何轻松地在服务端渲染 React 组件。

8.2　使用服务端渲染的原因

服务端渲染是一项很棒的特性，但不能为了用它而用它，而应该有真正充分的使用理由。在本节中，我们将探究服务端渲染如何帮助提升应用性能，以及它能为我们解决哪些问题。

8.2.1　SEO

在服务端渲染应用的一个主要原因就是 SEO。

实际上，如果为主流搜索引擎的爬虫提供空壳 HTML，那么它们将无法从中解析出任何有意义的信息。

如今，Google 的爬虫似乎能执行 JavaScript 代码，但仍然有很多限制，而 SEO 往往对我们的业务起关键作用。

多年来，我们一直习惯于编写两个应用：一个在服务端渲染，供爬虫解析；另一个在客户端运行，供用户使用。

之所以这样做，是因为服务端渲染的应用无法满足用户期待的交互水准，同时客户端应用无法被搜索引擎索引。

维护和支持两个应用的难度很大，这导致代码库不够灵活，而且很难修改。

幸运的是，有了 React 后就可以在服务端渲染组件了。这样一来，爬虫就能轻易理解并索引我们提供的应用内容。

这不仅有利于 SEO，也便于我们使用社交分享服务。实际上，当在 Facebook 和 Twitter 这类平台分享页面时，它们允许我们定义分享出去的信息片段内容。

举例来说，可以通过 Open Graph 协议告诉 Facebook，我们希望在发布社交消息时显示某个特定页面上的某张图，并为这条消息指定特定标题。

纯客户端应用几乎无法实现这个需求，因为社交引擎从页面解析信息时用的是服务端返回的文档标记。

如果服务端对所有 URL 都返回空壳 HTML，那么在社交网络上分享页面时，Web 应用的消息片段也是空的，这将严重影响应用的推广。

8.2.2 通用代码库

我们没有太多可供选择的客户端技术，只能用 JavaScript 编写应用。一些语言可以在构建过程中转换为 JavaScript，但本质上还是一样。

对于可维护性和跨公司知识共享来说，在服务端和客户端使用同一种语言很有好处。

在客户端和服务端共享逻辑后，变更操作会变得更简单，不必再重复工作，错误和问题也大大减少。

比起更新两份应用代码，维护单个代码库的工作量要少得多。

考虑在服务端引入 JavaScript 的另一个理由是，前后端开发人员可以共享知识。

两端复用代码能够使合作变得更加方便，整个团队采用同种语言也有利于快速决策和修改。

8.2.3　性能更强

客户端应用深受我们喜爱，因为它们反应迅速而且具备响应式特性。然而，它们的问题在于，用户操作应用前，需要加载并运行文件包。

如果用户使用笔记本电脑或者台式机，并且网速很快，那么这不算什么问题。然而，如果在移动设备上通过 3G 网络加载超大的 JavaScript 文件包，那么用户要等待一小段时间才能操作应用。这不仅有损整体的用户体验，也会影响用户转化率。主流电商网站已经证实，页面加载时间仅增加几毫秒就会对营收造成巨大冲击。

举例来说，如果服务端托管的应用只有空的 HTML 页面和`<script>`标签，并且在用户可以点击页面前只显示加载动画，那么网站的感知速度将会受到严重影响。

相反，如果在服务端渲染网站，用户一访问页面就能看到部分内容，那么他们留下来的可能性会更高，尽管他们仍然需要等待同样久的时间才能进行实际操作，因为不管有没有服务端渲染，都需要加载客户端文件包。

可以用服务端渲染极大地提升感知性能，因为我们可以在服务端输出组件并直接为用户返回一些信息。

8.2.4　不要低估复杂度

显然，尽管 React 提供了简单的 API 用于在服务端渲染组件，但创建通用应用是有代价的。因此，即使有了上述理由，在启用前也应该充分考虑并弄清团队是否准备好支持和维护通用应用。

我们将在后面几节中了解到，创建服务端渲染的应用实际上要做的不只是渲染组件。

我们要搭建和维护带有路由和逻辑的服务器、管理服务端数据流等。如果有可能，还要缓存服务器内容，以便更快地输出页面。除此之外，维护功能完整的通用应用还有许多其他任务要完成。

出于以上原因，我的建议是先开发客户端版本，只有当 Web 应用能良好地在服务端运行时，才应该启用服务端渲染来改善体验。

只有真正需要时才应该启用服务端渲染。比如，需要 SEO 或者定制社交分享信息时，可以考虑使用服务端渲染。

如果发现加载整个应用耗时很久，而你已经采取了一切优化手段（下一章将详细讨论优化这一话题），那么可以考虑用服务端渲染提升感知速度，以便为用户提供更好的体验。

Facebook 的工程师 Christopher Pojer 曾在 Twitter 上提过，他们为 Instagram 启用服务端渲染

只是为了 SEO，因为对于 Instagram 这类拥有大量动态内容的网站来说，服务端渲染在提升感知速度方面没什么用。

8.3 基础示例

我们将创建一个非常简单的服务端应用，通过它来了解构建通用应用的基本步骤。

我们有意精简了这个构建示例，因为本节的目的是展示服务端渲染的工作原理，而不是提供详细的解决方案或代码模板。即便如此，你仍然可以在开发真正的应用时将该示例应用作为基础。

> 本节假设你熟悉与 JavaScript 构建工具相关的所有概念（如 Webpack 及其加载器），并且对 Node.js 有基本了解。即使从未接触过 Node.js 应用，JavaScript 开发人员也应该能够轻松理解本节内容。

示例应用将包含以下两个部分。

- 服务端：我们将用 Express 搭建基础的 Web 服务器，它负责托管服务端渲染的 React 应用的 HTML 页面。
- 客户端：我们将照常用 `react-dom` 渲染应用。

应用两端的代码都会用 Babel 进行转译，运行前还会用 Webpack 打包，这样我们就能充分利用 ES2015 的能力，并在 Node.js 和浏览器环境下使用模块。

首先，进入一个空文件夹，执行以下命令来创建新的应用包：

```
npm init
```

生成 package.json 文件后，安装依赖。需要先安装 Webpack：

```
npm install --save-dev webpack
```

接着安装 Babel 的加载器，以及用 React 和 JSX 编写 ES2015 应用时所需要的预设配置：

```
npm install --save-dev babel-loader babel-core babel-preset-es2015 babel-preset-react
```

创建服务端打包文件还需要一项依赖。Webpack 允许我们定义一系列外部依赖，即不需要将这些依赖加入打包文件。在构建服务端应用时，实际上不需要添加开发时用到的所有节点包；我们只想打包服务端代码。有一个包可以帮助我们做到这一点，将它用到 Webpack 配置的外部入口部分就可以排除所有模块：

```
npm install --save-dev webpack-node-externals
```

非常好，现在可以在 package.json 的脚本部分添加一条入口命令，然后在终端中简单地执行 `build` 命令即可：

```
"scripts": {
  "build": "webpack"
},
```

然后创建配置文件 webpack.config.js，它负责告诉 Webpack 如何打包文件。

先导入设置外部节点依赖的库。另外还要定义客户端和服务端都要用到的Babel加载器的配置：

```
const nodeExternals = require('webpack-node-externals')

const loaders = [{
  test: /\.js$/,
  exclude: /(node_modules|bower_components)/,

  loader: 'babel',
  query: {
    presets: ['es2015', 'react'],
  },
}]
```

我们在第 7 章中学习过如何从配置文件中导出配置对象。Webpack 提供了一项很棒的特性，这个特性允许我们导出配置对象数组，这样就可以在同一个文件中定义客户端和服务端的配置，一步到位地使用它们。

我们应该很熟悉客户端配置了：

```
const client = {
  entry: './src/client.js',

  output: {
    path: './dist/public',
    filename: 'bundle.js',
  },

  module: { loaders },
}
```

我们告诉 Webpack，客户端应用的源代码位于 src 文件夹，并指定在 dist 文件夹中生成打包文件。

接着，将刚刚创建的 `babel-loader` 的配置对象设置给模块部分的 `loaders` 属性。这个步骤非常简单。

服务端配置稍微有些不同，但你应该能轻松地掌握并理解：

```
const server = {
  entry: './src/server.js',

  output: {
    path: './dist',
    filename: 'server.js',
```

```
  },

  module: { loaders },

  target: 'node',

  externals: [nodeExternals()],
}
```

如你所见，入口、输出以及模块部分基本一样，只不过文件名有些区别。

服务端配置中新增了 `target` 部分，我们指定其值为 `node`，以此告诉 Webpack 忽略 Node.js 的所有内置系统包，如 `fs`。另外，`externals` 部分用之前导入的库告诉 Webpack 忽略这些依赖。

最后，还需要以数组的形式导出这两个配置对象：

```
module.exports = [client, server]
```

配置完毕。现在可以开始编写代码了，先从我们更熟悉的 React 应用入手。

创建 src 文件夹，并在其中创建 app.js 文件。

app.js 包含以下内容：

```
import React from 'react'

const App = () => <div>Hello React</div>

export default App
```

代码很简单：导入 React，创建渲染出 Hello React 消息的 `App` 组件，最后导出该组件。

现在创建 client.js，它负责在 DOM 中渲染 `App` 组件：

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './app'

ReactDOM.render(<App />, document.getElementById('app'))
```

当然，我们很熟悉这一步。导入 React、ReactDOM 以及上一步中创建的 `App` 组件，然后用 ReactDOM 在 ID 为 `app` 的 DOM 元素内渲染该组件。

接下来开始实现服务端部分。

首先，创建 template.js 文件，用它导出的函数返回页面标记，再由服务器将该标记发送给浏览器：

```
export default body => `
  <!DOCTYPE html>
```

8

```
<html>
  <head>
    <meta charset="UTF-8">
  </head>
  <body>
    <div id="app">${body}</div>
    <script src="/bundle.js"></script>
  </body>
</html> `
```

这段代码很简单：函数接受 body 参数（后面会介绍这个参数包含 React 应用），并返回页面结构。

值得一提的是，即使应用在服务端渲染，客户端也需要加载文件包。其实，服务端渲染只是 React 渲染工作的一半。我们仍然想要一个客户端应用，以便使用浏览器的所有特性，比如事件处理器。

现在是时候创建 server.js 了，它的依赖更多，因此值得详细探究一番：

```
import express from 'express'
import React from 'react'
import ReactDOM from 'react-dom/server'
import App from './app'
import template from './template'
```

首先导入 express，它可以很简单地创建带路由的 Web 服务器，也可以托管静态文件。

接着导入 React、ReactDOM 以及需要渲染的 App 组件。注意 ReactDOM 导入语句中的 /server 路径。最后，导入刚刚定义的模板。

现在可以创建 Express 应用了：

```
const app = express()
```

告诉应用静态资源的存储路径：

```
app.use(express.static('dist/public'))
```

你可能已经注意到了，这里的路径就是 Webpack 客户端配置中输出打包文件的目标路径。

接下来就是用 React 进行服务端渲染的逻辑部分：

```
app.get('/', (req, res) => {
  const body = ReactDOM.renderToString(<App />)
  const html = template(body)
  res.send(html)
})
```

这里告诉 Express，我们想要监听路由 /，当客户端访问这条路由时，用 ReactDOM 库将 App 组件渲染成字符串。此处体现了 React 的服务端渲染的神奇和简约。

renderToString 方法返回 App 组件生成的 DOM 元素的字符串形式，如果使用 ReactDOM 的 render 方法，那么它会在 DOM 中渲染出一模一样的树状结构。

body 变量的值如下所示：

```
<div data-reactroot="" data-reactid="1" data-react-
checksum="982061917">Hello React</div>
```

如你所见，body 的值表示 App 组件的 render 方法中定义的内容，只不过多出来几个 data 属性，React 要在客户端用这些属性将应用附加到服务端渲染的字符串中。

现在我们有了应用的服务端渲染形式，可以用 template 函数将它插入 HTML 模板，并通过 Express 响应返回给浏览器。

最后，启动 Express 应用：

```
app.listen(3000, () => {
  console.log('Listening on port 3000')
})
```

只剩少数几个步骤，一切就准备就绪了。

第一步，定义 npm 的启动脚本，用它启动节点服务器：

```
"scripts": {
  "build": "webpack",
  "start": "node ./dist/server"
},
```

准备好脚本后，先执行以下命令来构建应用：

```
npm run build
```


打包完成后，再执行以下命令：

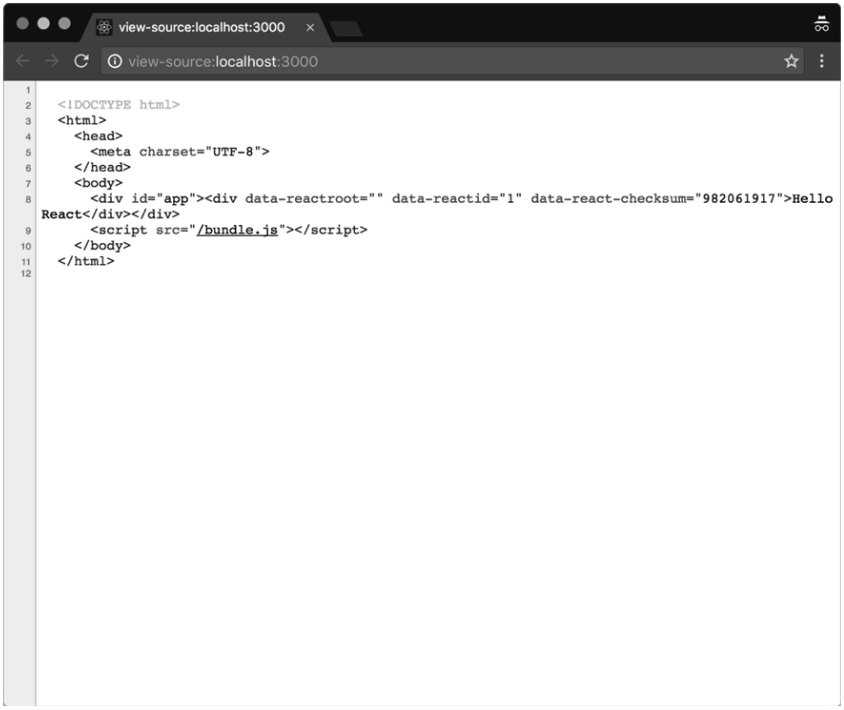
```
npm start
```

在浏览器中打开 http://localhost:3000，并查看结果。

需要重点关注两点。首先，如果使用浏览器的**查看网页源代码**功能，我们可以看到服务端返回的应用渲染完成后的源代码，但没有启用服务端渲染的情况下是无法查看的。

其次，如果在已安装 React 插件的情况下打开开发者工具，我们就会看到客户端也渲染了 App 组件。

以下截图展示了网页源代码。



8.4　数据获取示例

上一节中的示例已经清楚解释了如何搭建 React 通用应用。

这非常简单明了，而且主要侧重于如何通过配置来完成任务。

但真实的应用需要加载数据，而不仅仅是示例中的 `App` 组件这类静态 React 组件。假设我们想在服务端加载 Dan Abramov 的 gist，并通过刚刚创建的 Express 应用返回列表数据。

我们通过第 5 章中的数据获取示例了解了如何用 `componentDidMount` 触发数据加载。这种做法并不适合服务端，因为组件没有挂载到 DOM 上，也就永远不会触发这个生命周期钩子。

用 `componentWillMount` 这种更早执行的钩子也行不通，因为数据获取操作是**异步的**，但 `renderToString` 不是。因此，我们需要找到一种方法来预先加载数据，并将它作为 props 传递给组件。

我们来看看如何稍微修改一下上一节中的应用，使其可以在服务端渲染阶段加载 gist。

首先，修改 app.js 接收 gist 列表作为 prop，然后在 `render` 方法中遍历它，以显示每一项的描述：

```
const App = ({ gists }) => (
  <ul>
    {gists.map(gist => (
      <li key={gist.id}>{gist.description}</li>
    ))}
  </ul>
)

App.propTypes = {
  gists: React.PropTypes.array,
}
```

根据前面学到的概念，我们定义了一个无状态函数式组件，它接收 `gists` 作为 prop 并遍历元素来渲染列表项。

现在我们需要修改服务端代码，以便获取 `gists` 并传递给组件。

要想在服务端使用**获取** API，需要先安装 `isomorphic-fetch` 库，该库按照标准实现了获取函数。Node.js 和浏览器环境中都能使用它：

```
npm install --save isomorphic-fetch
```

先在 server.js 中导入这个库：

```
import fetch from 'isomorphic-fetch'
```

调用 API 的代码如下所示：

```
fetch('https://api.github.com/users/gaearon/gists')
  .then(response => response.json())
  .then(gists => {

  })
```


此处可以在最后一个 `then` 方法内使用 `gists` 变量。我们的示例要将它传递给 `App` 组件。

因此，可以将/路由的代码改写为以下样子：

```
app.get('/', (req, res) => {
  fetch('https://api.github.com/users/gaearon/gists')
    .then(response => response.json())
    .then(gists => {
      const body = ReactDOM.renderToString(<App gists={gists} />)
      const html = template(body)

      res.send(html)
    })
})
```

此处先获取 `gists`，然后传给 `App` 组件的属性，再将它渲染成字符串。

一旦 `App` 组件完成渲染，我们就能得到其标记，并用前一节中的 `template` 函数将它返回

到浏览器。

在控制台中执行以下命令，并在浏览器中访问 http://localhost:3000。你应该可以看到服务端渲染的 gists 列表：

```
npm run build && npm start
```

为了确保列表是 Express 应用渲染的，可以访问：

```
view-source:http://localhost:3000/
```

你会看到页面标记以及 gists 的各项描述。

一切都很完美，而且步骤很简单，但如果打开开发者工具控制台，我们会看到以下错误信息：

```
Cannot read property 'map' of undefined
```

出现这个错误的原因在于，客户端会再次渲染 App 组件，但此时没有传递任何 gists 给它。

这一开始听起来有些违反直觉，因为我们认为 React 十分智能，能够在客户端使用服务端字符串中渲染的 gist 数据。但事实并非如此，因此我们要寻找一种方法，以便客户端也能获取 gist。

你可能想到在客户端再次执行获取。这样的确可行，但不太理想，因为最终要触发两次 HTTP 请求：Express 服务器一次，浏览器一次。

仔细想想，我们已经在服务端发起过请求，并获取了所需的一切数据。在服务端和客户端之间共享数据的典型方案是，从 HTML 标记中脱离数据，再注回浏览器。

这个概念似乎很复杂，其实不然。现在我们就来看看它实现起来有多简单。

首先，从服务端取回 gist 后，必须将它们注入模板。实现这一点需要稍微修改一下模板：

```
export default (body, gists) => `
  <!DOCTYPE html>
  <html>
    <head>
      <meta charset="UTF-8">
    </head>
    <body>
      <div id="app">${body}</div>
      <script>window.gists = ${JSON.stringify(gists)}</script>
      <script src="/bundle.js"></script>
    </body>
  </html>
`
```

template 函数现在接收两个参数：应用的 body 部分和 gists 集合。

前者插入 id 为 app 的元素，后者用于定义 window 对象上的全局 gists 变量，这样我们就

可以在客户端使用它了。

在 Express 路由代码中，只需要修改传递 body 生成模板的那行代码，如下所示：

```
const html = template(body, gists)
```

最后，我们需要在 client.js 中使用 window 对象上的 gists 属性，这也非常简单：

```
ReactDOM.render(
  <App gists={window.gists} />,
  document.getElementById('app')
)
```

我们直接读取 gists 属性，并将它们传递给客户端渲染的 App 组件。

现在再次执行以下命令：

```
npm run build && npm start
```

在浏览器窗口中打开 http://localhost:3000，现在不会再出现错误信息了，而如果用 React 开发者工具查看 App 组件，我们就能看到它在客户端接收了 gists 集合。

8.5 Next.js

你已经见识了 React 服务端渲染的基本概念，也可以用刚刚创建的项目作为真正应用的基础。

然而，你可能觉得模板代码太多，还要了解许多不同工具才能运行一个简单的 React 通用应用。

这种感觉很普遍，称为 JavaScript 疲劳，本书一开始介绍过。

好在 Facebook 的开发人员和 React 社区中的其他公司都非常努力地改进开发体验，以便开发人员的生活可以更轻松。此时，你可能已经用过 create-react-app 来尝试前面几章中的示例，也应该能体会到它可以非常方便地创建并运行 React 应用，且开发人员无须学习大量技术和工具。

现在的 create-react-app 还不支持服务端渲染，但 Zeit 公司开发了一款名为 Next.js 的工具，它可以极其方便地生成通用应用，无须关心配置文件。此外，它还大大减少了模板代码。

值得一提的是，抽象始终有利于快速构建应用。但关键在于，添加太多层抽象前应该理解内部原理，这也正是为何学习 Next.js 之前我们需要先手动实现一遍。

我们已经了解了服务端渲染的工作原理，并学会了如何将状态从服务端传到客户端。理解基本概念后，可以选用一项工具将复杂之处隐藏起来，以便我们只写少量代码就可以实现同样的目的。

接下来我们将创建和刚才一样加载 Dan Abramov 的所有 gist 的应用，你会看到有了 Next.js 之后代码变得多么简单清晰。

首先，进入空文件夹并创建一个新项目：

```
npm init
```

然后安装 Next.js 库：

```
npm install --save next
```

现在项目创建完毕，只要添加一条 npm 脚本就能运行这个库：

```
"scripts": {
  "dev": "next"
},
```

非常完美，现在开始生成 App 组件。

Next.js 依赖于约定，其中最关键的一条约定是，创建的页面要和浏览器 URL 匹配。默认页面是 index，因此我们创建一个名为 pages 的文件夹，并在其中创建 index.js 文件。

导入依赖：

```
import React from 'react'
import fetch from 'isomorphic-fetch'
```

这里再次导入了 isomorphic-fetch 库，因为我们想要在服务端使用获取函数。

接着定义一个名为 App 的类，它继承自 React.Component：

```
class App extends React.Component
```

在该类中定义一个带有 static 和 async 关键词的方法 getInitialProps，在这个方法中告诉 Next.js 我们想要在服务端和客户端加载什么数据。库会使该方法返回的对象可以被组件作为 props 使用。

类方法带有 static 和 async 关键词表示这个方法可以被类实例的外部访问，并且它会逐步执行自身内部的等待指令（await 关键词）。

这些概念非常高级，不在本章的范畴之内，如果对它们感兴趣，你可以查看 ECMAScript 提案。

我们刚刚讨论的方法的具体实现如下所示：

```
static async getInitialProps() {
  const url = 'https://api.github.com/users/gaearon/gists'
  const response = await fetch(url)
  const gists = await response.json()

  return { gists }
}
```

我们告诉该方法发起获取请求并等待响应；接着将响应转换为 JSON，这个过程会返回一个

promise 对象。promise 变成已完成状态后，就可以返回包含 `gists` 属性的 props 对象了。

组件的渲染方法和之前很相似：

```
render() {
  return (
    <ul>
      {this.props.gists.map(gist => (
        <li key={gist.id}>{gist.description}</li>
      ))}
    </ul>
  )
}
```

不过，它要通过 `this.props.gists` 访问数据，因为我们现在位于类实例内部。

最后，定义 `PropTypes`，创建组件时始终要记得这一步：

```
App.propTypes = {
  gists: React.PropTypes.array,
}
```

然后导出组件：

```
export default App
```

现在打开控制台，执行以下命令：

```
npm run dev
```

我们将看到以下输出：

```
> Ready on http://localhost:3000
```

在浏览器中打开这个 URL，就会看到通用应用已经运行起来。

有了 Next.js，只要几行代码，无须任何配置就能很便捷地搭建一个通用应用，这实在令人印象深刻。

你可能也注意到了，如果在编辑器内修改应用，不用刷新页面就能立刻在浏览器中查看结果。这是 Next.js 的另一项特性——热模块替换。它在开发模式下非常有用。

如果喜欢本章，可前往 GitHub 为 Next.js 加一颗星。

8.6 小结

服务端渲染的旅程已经告一段落。现在你应该学会了如何创建 React 服务端渲染应用，而且应该清楚地了解它为什么对你有用。SEO 显然是最主要的理由之一，但社交分享和性能同样也是

很重要的因素。

你学习了如何在服务端加载数据，并从 HTML 模板中脱离数据，以便客户端应用在浏览器中运行时能够访问数据。

最后，你看到了 Next.js 这类工具如何帮助减少模板代码，并隐藏搭建 React 服务端渲染应用给代码库造成的复杂度。

下一章将探讨与性能相关的话题。

第 9 章 提升应用性能

Web 应用的高性能是提供良好用户体验与提升用户转化率的关键。React 通过不同的技巧快速渲染组件，并尽量减少操作 DOM。因为修改 DOM 的性能开销往往很大，所以减少操作次数非常关键。

但在某些特定场景下，React 无法优化运行过程，这就要靠开发人员用一些特殊方案使得应用流畅运行。

在本章中，我们将学习 React 性能方面的基础概念，以及如何使用一些 API 帮助 React 在不损伤用户体验的情况下最优地更新 DOM。另外，我们还会看到一些不利于应用运行，甚至会导致应用运行缓慢的常见错误。

通过本章中的几个简单示例，你会接触到监控代码库性能和定位瓶颈的工具。我们还会学习为何不可变性与 `PureComponent` 是构建高性能 React 应用的最佳工具。

我们不应该盲目地优化组件，本章介绍的优化技巧只应该在必要时使用。

本章包含如下内容。

- 一致性比较的原理，以及如何使用 `key` 属性帮助 React 更好地工作。
- 如何使用生产版本的 React 以便更快运行。
- `shouldComponentUpdate` 和 `PureComponent` 能为我们做什么，以及如何使用它们。
- 常用的优化手段以及与性能相关的常见错误。
- 不可变数据的含义与用法。
- 促使应用更快运行的工具与库。

9.1 一致性比较与 key 属性

大多数情况下，React 默认已经够快了，我们不需要多做什么来提升应用性能。React 用了许多不同技巧来优化组件在屏幕上的渲染过程。

当显示组件时，React 会调用自己的**渲染方法**，还会递归调用子组件的渲染方法。组件的渲染方法会返回 React 元素树，然后 React 根据它来判断更新 UI 需要执行哪些 DOM 操作。

当组件的状态发生变化时，React 会再次调用该节点的渲染方法，并将渲染结果与之前的 React 元素树进行比较。库本身非常智能，能够计算出使屏幕上产生预期变化所需的最小操作集合。这个过程称为**一致性比较**，并由 React 透明地管理。正因为这一点，我们才能简单地声明式描述某个特定时刻的组件样子，并将其余的工作交给库。

React 尝试尽可能少地操作 DOM，因为直接操作**文档对象模型**的性能开销非常大。

然而，比较两棵元素树并非没有开销，因此 React 通过两项设定降低了其中的复杂度：

- 如果两个元素的类型不同，则它们渲染的树也不同；
- 开发人员可以用 `key` 属性标记子组件，使它们在不同渲染过程得以保留。

从开发人员的视角来看，第二点非常有趣，因为它提供了一项工具来帮助 React 更快地渲染视图。

接下来我们将通过一个简单示例来解释如何恰当用 `key` 显著提升性能。

我们从创建一个简单组件和一个按钮开始，前者用于展示项目列表，后者用于向列表添加项目并导致组件重新渲染。

这里创建了一个类，因为我们想要在状态中保存当前列表，并且需要用事件处理器监听按钮的点击事件：

```
class List extends React.Component
```

在 `List` 组件的构造器中初始化列表，并绑定事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
  }

  this.handleClick = this.handleClick.bind(this)
}
```

事件处理器会在列表中新增一项，并将最终的数组保存在状态中。

```
handleClick() {
  this.setState({
    items: this.state.items.concat('baz'),
  })
}
```

最后，render 方法会遍历状态中的 items 属性，显示列表的每个元素，同时声明 button 元素以及相应的 onClick 事件处理器。

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => <li>{item}</li>)}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

组件已经就绪，如果将它添加到应用中（或者用 create-react-app 创建新应用来尝试），你就能看到屏幕上显示了 foo 和 bar 这两项，点击+按钮会在列表底部新增 baz。

这正是我们想要的结果，但为了清楚地了解 React 的行为，需要安装一个新的工具来帮助我们记录和显示与性能相关的信息。这个工具是一个 React 插件，执行以下命令即可安装：

```
npm install --save-dev react-addons-perf
```

安装完成后，将它导入前面的 List 组件：

```
import Perf from 'react-addons-perf'
```

Perf 对象提供了一些有用的方法来监控 React 组件的性能。使用 start()可以开始记录数据，使用 stop()可以告诉插件我们已经收集了足够多的数据并准备显示出来。

还有很多不同的方法可以用来显示浏览器控制台中收集到的数据。其中 printWasted 用处最大，它可以打印出组件执行渲染方法所耗费的时间，并返回上次渲染过程中的相同元素。Perf 插件还提供了 DOM 中与 React 操作相关的一些有用信息，对应的方法名为 printOperations，接下来我们就会用它来验证至关重要的 key 属性如何提升应用的性能。一旦组件完成更新，需要显示浏览器中的 DOM 操作的结果时，这个方法就会被调用。我们可以简单地用 React 提供的两个生命周期钩子来开启或关闭数据记录功能，并显示结果。

我们要用到的第一个方法是 componentWillUpdate，组件将要更新和重渲染前会立刻触发它：

```
componentWillUpdate() {
  Perf.start()
}
```

我们在这个生命周期钩子内调用 Perf 插件的 start()方法来开始监控性能。一旦更新完成，就在 componentDidUpdate 中停止记录，如下所示：

```
componentDidUpdate() {
  Perf.stop()
  Perf.printOperations()
}
```

如你所见，此处除了停止记录，还调用了 `Perf` 插件的 `printOperations` 方法，这样就能看到屏幕上添加 baz 元素时 React 具体对 DOM 做了什么操作。

如果运行组件代码并点击+按钮，开发者工具控制台就会列出具体操作的表格。表格中对我们有用的两列是 Operation 和 Payload，前者显示 `"insert child"`，后者显示 `"{"toIndex":2, "content":"LI"}"`。

React 会判断出，在当前列表末尾添加子项这一改动实际上要在列表编号为 2 的位置（第 3 个元素）插入新的子元素 `LI`。

你可能已经注意到了，React 没有重新绘制整个屏幕，而是计算了更新 DOM 所需的最少操作数。这种做法非常巧妙，而且能够满足大多数用例。

然而，React 在一些特殊场景下不够智能，无法判断更新 DOM 的最优路径，因此我们要用 `key` 属性来帮助它。如果稍微修改一下前面的示例，特别是事件处理器的部分，不再将 baz 加到列表末尾，而是将它插到最前面，然后来看看 React 行为的缺陷。

复制保存在当前状态中的数组，并用 JavaScript 的 `unshift` 方法将一个元素插到副本数组的第一位置。在副本数组上进行操作的原因是，`unshift` 方法不会返回新数组，而是修改原数组，在操作状态时应当极力避免这一点。新的 `onClick` 处理器代码如下所示：

```
handleClick() {
  const items = this.state.items.slice()
  items.unshift('baz')

  this.setState({
    items,
  })
}
```

上述代码复制了原数组，然后在顶部插入 baz 元素，最后将新数组赋值给状态，这就触发了重新渲染。

如果运行 `List` 组件，就可以在屏幕上看到最初的列表项 foo 和 bar，点击+按钮，新的 baz 元素会成为第一个元素。

一切都和预期一样，但如果再次查看浏览器开发者工具就会发现，React 执行了多次操作。从前面我们最感兴趣的 Operation 和 Payload 这两列可以得出，React 进行了三步改动：

- 将列表第一项的文本替换为新值 baz；
- 将列表第二项的文本替换为之前第一项的值 foo；

- 在编号为 2 的位置（即列表底部）插入新的子元素。

React 没有直接在列表顶部添加新元素并将原有元素向下移，而是修改了原有元素的值并在列表底部添加了新元素。

React 这么做的原因在于，它只是检查前后两次子元素是否一致，如果发现第一个元素不同，那么它会认为所有列表项都是新的，因此全部修改。

这个简单示例没有展示任何可见的性能问题，但如果是包含上百个元素的真实应用，那么问题就很严重了。

好在 React 提供了 `key` 属性这一工具，我们可以用它帮助 React 判断哪些元素进行了修改，以及新增或移除了哪些元素。

`key` 的使用很简单，为列表中的每一项添加唯一的 `key` 属性即可。此处重点在于，`key` 属性的值在每次渲染过程中不能改变，因为 React 会比较渲染前后的 `key` 属性值，从而判断元素是新增的还是已有的。

举例来说，我们可以修改 `List` 组件的渲染方法，如下所示：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => <li key={item}>{item}</li>)}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

将每个列表项的 `key` 属性值设为该项本身，然后在浏览器中再次运行组件，我们会发现效果还是没有变化：先显示两个列表项，点击按钮后会在顶部新增一项。

然而，如果打开开发者工具，我们会看到 `Perf` 插件打印的信息和上次不同。

事实上，Operation 显示刚刚插入了单个元素，更重要的是，Payload 显示元素放到了编号为 0 的位置，也就是第一位。

这很巧妙：我们用 `key` 帮助 React 判断出操作的最少集合，从而提升组件的渲染性能。牢记这条简单规则，就能解决大量因为没有使用 `key` 而造成性能损耗的案例。

值得一提的是，记住这条规则很容易，因为每当我们忘记添加 `key` 属性时，React 就会在浏览器控制台给出以下警告：

```
Each child in an array or iterator should have a unique "key" prop.Check the render
method of `List`.
```

这条错误消息非常有用，因为它告诉我们修改哪个组件才能解决问题。

另外，如果你按照第 2 章中那样启用 Eslint，并开启 `eslint-plugin-react` 插件的 `jsx-key` 规则，那么 linter 也会在缺少 `key` 属性时给出警告。

9.2　优化手段

注意，不管是用 `create-react-app` 创建的还是从头编写的，本书所有示例中的应用使用的都是开发版的 React。

开发版的 React 对编码和调试过程很有帮助，因为它可以提供解决各种问题所需的必要信息。但这些校验和警告都会损耗性能，生产环境下应当移除。

因此，应用首先要优化的地方就是，构建打包时要将 `NODE_ENV` 环境变量设置为 `production`。Webpack 很容易做到这一点，使用 `DefinePlugin` 插件即可，如下所示：

```
new webpack.DefinePlugin({
  'process.env': {
    NODE_ENV: JSON.stringify('production')
  }
}),
```

为了获得最优性能，不仅构建打包时需要启用生产环境标记，还要压缩最终代码来减小体积，以便应用可以更快加载。要想实现这一点，只要简单地在 Webpack 配置的插件列表加入以下插件：

```
new webpack.optimize.UglifyJsPlugin()
```

如果启用生产版本的 React 后仍然发现应用有些地方运行得很慢，还可以采取一些技巧来改进组件的渲染过程。

这里重点要提的是，没有测试应用性能并弄清瓶颈所在前，不应该优化应用。过早优化往往会带来不必要的复杂度，应该极力避免这种情况。

另外还要注意一点，React 本身已经采取了一些措施使应用运行得更快、更流畅，因此大部分情况下不需要我们再多做什么。

然而，这些优化手段无法满足一些特殊场景，我们要帮助库尽量提供最佳体验。在这些场景下，我们要告诉 React 不要比较元素树某些部分的一致性。

9.2.1　是否要更新组件

比较器算法的原理常常让许多开发人员感到困惑。他们往往认为，既然 React 足够智能，能

够寻找修改 DOM 的最快路径，那么组件没有发生变化就不会触发渲染方法。但事实并非如此。

实际上，为了找出减少 DOM 操作的必要步骤，React 必须触发所有组件的渲染方法，并比较前后两次的结果。

如果比较后发现什么都没有改变，那么就不修改 DOM，这是理想情况。但如果渲染方法执行了很复杂的操作，React 就要耗费一些时间才能判断出不需要进行任何修改，这种情况就不够理想了。

很显然，我们希望组件尽可能简单，还应该避免在渲染方法中执行开销很大的操作。然而，有时不太好掌控这一点，因此应用就会运行缓慢，尽管我们从未操作 DOM。

React 无法判断哪个组件不需要更新，但我们可以调用一个方法告诉它更新组件的时机。

这个方法就是 `shouldComponentUpdate`，如果它返回 `false`，那么在父组件的更新过程中，组件及其全部子元素的渲染方法不会被调用。

举例来说，为前面创建的列表示例添加以下代码：

```
shouldComponentUpdate() {
  return false
}
```

此时你会发现无论如何点击+按钮，浏览器中的应用都不会有任何效果，尽管状态确实发生了变化。这是因为我们告诉 React 不需要更新该组件。

总是返回 `false` 并没有什么实际用处，因此开发人员往往会在该方法中检查 props 或者状态是否改变。

以 `List` 组件为例，我们应该检查 `items` 数组是否改动过，并返回相应值。

`shouldComponentUpdate` 方法会传入两个参数来实现上述检查：两者分别是 `nextProps` 和 `nextState`。

拿本例来说，我们可以编写以下代码：

```
shouldComponentUpdate(nextProps, nextState) {
  return this.state.items !== nextState.items
}
```

只有 `items` 数组发生变化时，该方法才会返回 `true`，其他情况则不会进行渲染。举例来说，假设 `List` 组件的父组件更新非常频繁，但父组件状态又不会影响 `List`，此时就可以用 `shouldComponentUpdate` 告诉 React 不要调用 `List` 组件及其子组件的渲染方法。

检查所有 props 和状态属性是否变化非常繁琐枯燥，而且有时很难维护复杂的 `shouldComponentUpdate` 方法实现，尤其是需求频繁变动时。

出于以上原因，React 提供了一个特殊组件供我们继承使用，它可以浅比较所有 props 和状态属性。

它的用法很直观，创建组件类时继承 `React.PureComponent` 来代替 `React.Component` 即可。

要重点注意的是，顾名思义，浅比较不会检查对象中嵌套的深层属性，并且有时会给出意外结果。

浅比较和不可变数据结构搭配使用效果非常好，本章后面会详细解释这点。另外要提一点，深度比较复杂对象所需的开销有时比渲染方法本身更大。

这也是为何只应在必要时使用 `PureComponent`，而且只能在测试完性能后，弄清哪个组件运行耗时太长后才可以使用它。

9.2.2　无状态函数式组件

有时让新手觉得不合常理的另一个概念是，无状态组件实际上不会带来任何性能上的提升。

我们已经在第 3 章中了解过，无状态组件的优点很多。尽管它们可以使应用更加简洁、更易分析，但（目前）没有任何内部实现能使它们更快运行。

我们很容易想当然地认为这类组件渲染起来应该更快，因为它们没有实例、状态和事件处理器，但目前情况并非如此。

或许它们将来会得到优化（取决于 React 团队），但现在其性能甚至更差，因为我们已经知道无法使用 `shouldComponentUpdate` 方法，也就不能帮助 React 更快地渲染元素树。

9.3　常用解决方案

我们已经知道可以用 `PureComponent` 告诉 React 渲染**子树**的时机。正确使用它就能极大提升应用性能。还要重点强调的是，只有检查过应用性能并找到瓶颈所在后，才可以使用它。

有些情况较为复杂，继承 `PureComponent` 并不能实现我们想要的结果。这往往是因为我们认为 props 或状态没有改变，但实际上它们的确变化了。有时还很难判断哪个 prop 导致了组件重新渲染，或者哪个组件可以用 `PureComponent` 优化。

大多数情况下，重构组件并正确使用状态对于优化应用有极大帮助。

我们将介绍一些常用工具和解决方案来解决重渲染问题，并找出哪些组件可以进行优化。另外，我们还将学习如何重构复杂组件，将它们拆分成小型组件以获得更好的性能。

9.3.1 why-did-you-update

还可以用其他方法找出不需要更新的组件。最简单的方法就是安装一个能自动提供信息的第三方库。

首先，在终端中输入以下命令：

```
npm install --save-dev why-did-you-update
```

然后在 React 的 `import` 语句后面添加以下代码：

```
if (process.env.NODE_ENV !== 'production') {
  const { whyDidYouUpdate } = require('why-did-you-update')
  whyDidYouUpdate(React)
}
```

这段代码的基本含义是，只在开发模式下加载库，并用 `whyDidYouUpdate` 方法封装 React。注意，生产环境中不要启用这个库。

现在我们回到上一节中的 `List` 组件示例并对其稍作修改，然后看看这个库的实际运用。

首先，修改 `render` 方法，如下所示：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item key={item} item={item} />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```


这里 `map` 方法内部不再渲染一个个单独的列表项，而是返回自定义的 `Item` 组件，我们向该组件传入当前列表项数据，并用 `key` 属性告诉 React 哪些组件在更新前就已经存在了。

现在继承 `React.Component`，创建 `item` 组件：

```
class Item extends React.Component
```

目前它只实现了 `render` 方法来负责 `List` 组件的 `render` 方法的职责：

```
render() {
  return (
    <li>{this.props.item}</li>
  )
}
```

在浏览器中运行组件前，可能还想要替换 Perf 插件的方法，因为现在我们更关心渲染未变组件耗费的时间，而不是执行了哪些 DOM 操作：

```
componentDidUpdate() {
  Perf.stop()
  Perf.printWasted()
}
```

非常好！如果此时在浏览器中渲染组件，那么我们就能看到 foo 和 bar 两项，如果点击+按钮，开发者工具控制台就会出现两条提示。

第一条提示是 whyDidYouUpdate 方法输出的，它会告诉我们可以避免重复渲染哪些组件：

```
Item.props
  Value did not change. Avoidable re-render!
  before Object {item: "foo"}
  after  Object {item: "foo"}
Item.props
  Value did not change. Avoidable re-render!
  before Object {item: "bar"}
  after  Object {item: "bar"}
```

尽管 React 没有操作 foo 和 bar 的 DOM 节点，但实际上还是会用原来的 props 调用这两个组件的渲染方法，这就使得 React 多做了额外的工作。这类信息用处很大，但有时难以发现。

上述提示的下方就是 Perf 插件的输出内容，该插件计算了 props 未改变时触发 Item 组件的渲染方法所浪费的时间。

如此一来，这个问题就很好解决了，将 Item 组件继承语句中的 extends React.Component 改为以下语句即可：

```
class Item extends React.PureComponent
```

再次打开浏览器并运行应用，此时点击+按钮，控制台不会输出任何提示，这就意味着 props 未改变时不会渲染任何 Item 组件。

从感知性能角度来看，这个小示例改动前后没有很大差别，但如果是包含数百个列表项的大型应用，这么一点改动就能带来巨大收益。

9.3.2　在渲染方法中创建函数

接下来，我们像开发真实应用那样继续为 List 组件添加特性，看看某些情况下是否会破坏 PureComponent 的优势。

举例来说，我们要给每个列表项添加一个点击事件处理器，当点击某项时，将其值输出到控制台。

以上做法对于真实应用没有多大用处，但你可以只花少量精力，就轻松弄懂如何创建更复杂的操作，比如用列表项的数据显示新窗口。

添加上述特性需要改动两处，分别是 List 组件的 render 方法与 Item 组件的 render 方法。

先来修改前者：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
            key={item}
            item={item}
            onClick={() => console.log(item)}
          />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

Item 组件新增了 onClick prop，并为它赋值一个函数，调用该函数时会将当前项的数据输出到控制台。

为了运行这个函数，Item 组件也要添加相应的逻辑，只要将 onClick prop 赋值给<li>元素的 onClick 处理器即可：

```
render() {
  return (
    <li onClick={this.props.onClick}>
      {this.props.item}
    </li>
  )
}
```


这仍然是一个纯粹组件，列表中新增 baz 时，我们希望数据值没有变化的组件不要重新渲染。

问题是，如果在浏览器中运行组件，开发者工具会输出一些新的记录。首先，whyDidYou-Update 库会告诉我们 onClick 函数始终不变，这或许可以避免重新渲染：

```
Item.props.onClick
  Value is a function. Possibly avoidable re-render?
  before onClick() {
    return console.log(item);
  }
  after  onClick() {
    return console.log(item);
```

```
  }
Item.props.onClick
  Value is a function. Possibly avoidable re-render?
  before onClick() {
    return console.log(item);
  }
  after  onClick() {
    return console.log(item);
  }
```

其次，`Perf` 插件会提醒我们渲染 `List > Item` 组件浪费了一些时间。

React 认为每次渲染都传入了新函数的原因是，即使函数实现保持不变，每次调用箭头函数都会返回一个全新创建的函数。使用 React 常常会犯这个错，不过只要稍微重构一下组件就能轻松解决这个问题。

遗憾的是，因为需要知道对数函数由哪个子组件所触发，所以不能只在父组件中定义一次。因此，最好的做法似乎是在循环内部创建它。

实际上，我们要将部分逻辑移到 `item` 组件内，只有它才知道哪些列表项被点击了。

我们来看看继承自 `PureComponent` 的 `Item` 组件的完整实现，如下所示：

```
class Item extends React.PureComponent
```

在 `constructor` 方法中绑定点击事件的处理器：

```
constructor(props) {
  super(props)

  this.handleClick = this.handleClick.bind(this)
}
```

点击`<li>`元素时，`handleClick` 方法会调用从 `props` 接收到的 `onClick` 处理器，并传入当前列表项的数据。

```
handleClick() {
  this.props.onClick(this.props.item)
}
```

然后在 `render` 方法中使用刚刚创建的局部事件处理器：

```
render() {
  return (
    <li onClick={this.handleClick}>
      {this.props.item}
    </li>
  )
}
```

最后还要修改 `List` 组件的渲染方法，使其不要在每次渲染时返回新函数，如下所示：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item key={item} item={item} onClick={console.log} />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

如你所见，这里传递了我们需要用到的 `console.log` 函数，子组件会用正确的参数调用它。如此一来，`List` 组件内部的函数实例始终保持不变，也就不会触发任何重渲染过程。

如果再重新运行一遍组件，并点击+按钮来添加新项，`List` 组件会重新渲染，但此时没有浪费额外时间。

另外，如果点击列表中的任意一项，我们也可以看到控制台输出其值。

Dan Abramov 说过，在渲染方法中使用箭头函数（甚至在 `constructor` 方法中用 `bind` 绑定 `this`）的做法本身没什么问题，只不过实际使用 `PureComponent` 时要小心谨慎，确保不会引发不必要的重渲染。

9.3.3 props 常量

我们来继续改进列表示例，看看增加新特性会引发什么情况。

接下来我们将学习如何避免会导致 `PureComponent` 变得低效的一个常见错误用法。

假设 `Item` 组件需要新增一个 prop 来表示列表项所能拥有的一系列状态。实现这个需求的方式很多，而我们更关注如何传入默认值。

修改 `List` 组件的 `render` 方法，如下所示：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
            key={item}
            item={item}
            onClick={console.log}
            statuses={['open', 'close']}
          />
        ))}
```

```
        </ul>
        <button onClick={this.handleClick}>+</button>
      </div>
    )
  }
```

上述代码为每个 Item 组件设置了 key 和 item 这两个 prop，两者的值都是当前列表项 item。Item 组件内部触发 onClick 事件时会调用 console.log，这里还新增了一个 prop，用于表示列表项 item 可能拥有的状态。

现在，如果在浏览器中渲染 List 组件，foo 和 bar 会照常显示，点击+按钮新增 baz 时，我们会看到控制台输出一条新消息：

```
Item.props.statuses
  Value did not change. Avoidable re-render!
  before ["open", "close"]
  after  ["open", "close"]
Item.props.statuses
  Value did not change. Avoidable re-render!
  before ["open", "close"]
  after  ["open", "close"]
```

这条消息告诉我们，即使数组内部的值没有改变，但每次渲染都会给 Item 组件传入新的数组实例。

这种行为背后的原因是，所有对象在创建时都会返回新的实例，但即使包含相同的值，两个新数组也永远不会相等：

```
[] === []
false
```

控制台还打印了一张表格，以显示在不需要操作 DOM 时，React 用未变化的 props 渲染列表项所浪费的时间。

这个问题的解决方法就是只创建数组一次，每次渲染都传入相同的实例。

如下所示：

```
const statuses = ['open', 'close']
...
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => (
          <Item
            key={item}
            item={item}
            onClick={console.log}
            statuses={statuses}
```

```
          />
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

如果再次运行组件，我们可以看到现在控制台没有输出任何消息。这就表明新增元素时，列表项不会再多余地重新渲染。

9.3.4　重构与良好设计

接下来我们将学习如何重构已有组件（或者用更好的思路重新设计组件）来提升应用的性能。

选用糟糕的设计往往会带来很多问题。以 React 为例，如果将状态放在不合理的位置，那么组件就可能发生不必要的渲染。

之前提过，如果只是一个组件本身渲染多次，这并不算什么大问题。只有测试性能后，发现多次渲染很长的列表，进而导致应用响应性变差，问题才变得严重起来。

我们接下来要创建的组件和之前的很像，它类似于一个待办事项列表，还提供表单让用户输入新事项。

和之前一样，我们先从基础版本做起，再逐步优化。

该组件继承自 `React.Component`，类名为 `Todos`：

```
class Todos extends React.Component
```

在 `constructor` 方法中初始化状态并绑定事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
    value: '',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleClick = this.handleClick.bind(this)
}
```

状态包括以下两个属性。

- `items`：具有一些默认值的待办事项列表；新事项也会添加到该数组。
- `value`：用户填写新增事项的输入框的当前值。

现在你应该可以根据事件处理器的名称推断出其功能。每当用户在表单中输入字符时，就会触发 handleChange 方法：

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

正如我们在第 6 章中所学的那样，onChange 处理器接收带有目标属性的事件，其中目标属性表示输入元素，我们将它的值存储在状态中，以控制组件。

用户提交表单来新增事项时会触发 handleClick 方法：

```
handleClick() {
  const items = this.state.items.slice()
  items.unshift(this.state.value)

  this.setState({
    items,
  })
}
```

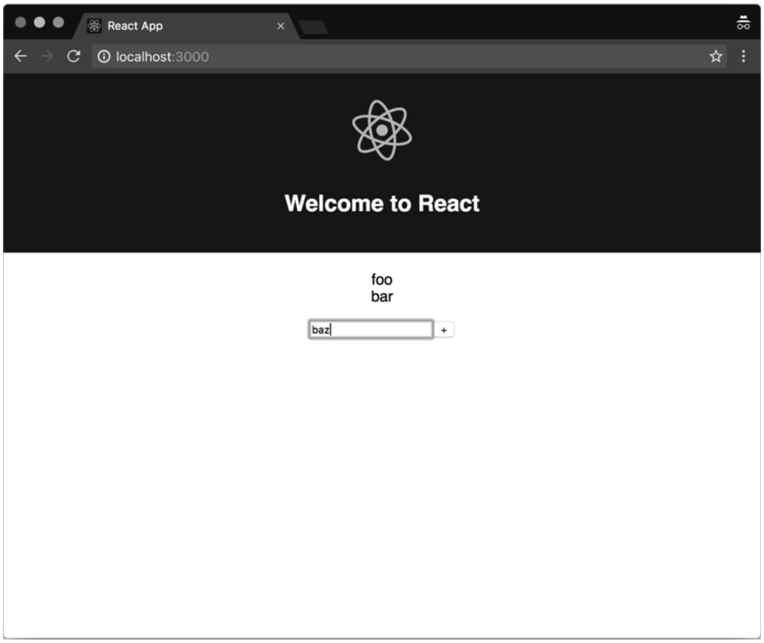
点击事件处理器和之前示例中的很像，只不过现在它使用了状态中的值，而不是字符串常量，而且它将值添加到了副本数组的头部。

最后，在 render 方法中编写视图：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map(item => <li key={item}>{item}</li>)}
      </ul>
      <div>
        <input
          type="text"
          value={this.state.value}
          onChange={this.handleChange}
        />
        <button onClick={this.handleClick}>+</button>
      </div>
    </div>
  )
```

这里我们声明了一个无序列表，在其中遍历事项集合，并用<li>元素输出每一项内容。列表下方是一个受控输入框，我们设置其当前值并监听 change 事件。最后还有一个+按钮，我们用它提交输入值并添加到列表数组。

以下是该组件的截图：



现在组件已经完成，在浏览器中运行它就能看到包含两个默认值的列表，以及一个用于新增列表项的表单。点击+按钮就能在列表顶部新增一项。

如果不给这个组件添加数百个列表项，就不会有什么特殊问题。但如果列表项过多，输入时就会感觉有些延迟。性能下降的原因是，列表项数量增多后，每次用户输入时都要渲染整个列表。当用受控输入框的新值更新状态时，React 会再次调用渲染方法来判断元素是否一致。

唯一改变的地方是 `input` 元素的值，只有这一部分需要修改 DOM；但为了找出必须执行的操作，React 会渲染整个组件以及所有子组件，多次重复渲染大型列表的性能开销非常大。

如果现在查看组件的状态对象，就可以清楚地发现其结构有问题。实际上，我们保存的列表项和表单值是两种完全不同的数据。

我们应该始终让组件做一件事情，而不是承担多个职责。

解决这个问题的方法是，将组件拆分成多个小组件，让它们各自负责简单的职责和状态。

由于小组件之间存在关联，我们还需要一个公有父组件。实际上，如果我们不想在每次用户输入时重新渲染列表，那么同样也不想在提交表单来新增事项时再次渲染列表。

为了实现这一点，我们将 `Todos` 组件修改为只负责保存 `items` 数组，列表和表单会共享这部分数据。

然后我们会创建独立的列表组件，它只接收 items 数组和表单组件，后者的状态用于控制输入框。提交表单时会触发公有父组件的回调函数，从而更新列表。

我们先来修改 Todos 组件：

```
class Todos extends React.Component
```

在 constructor 方法中，现在我们只在状态中定义了默认的 items 数组，并绑定了提交事件的处理器以作为 Form 组件的回调函数：

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
  }

  this.handleSubmit = this.handleSubmit.bind(this)
}
```

提交事件处理器的实现与前面点击事件的一样：

```
handleSubmit(value) {
  const items = this.state.items.slice()
  items.unshift(value)

  this.setState({
    items,
  })
}
```

不过这里会接收新列表项的值作为回调参数，然后复制数组并在头部插入新值。数组更新完毕后会被设置到状态中。

render 方法变得更加简洁，因为我们只要渲染两个自定义组件，再分别为它们传入 props：

```
render() {
  return (
    <div>
      <List items={this.state.items} />
      <Form onSubmit={this.handleSubmit} />
    </div>
  )
}
```

List 组件接收父组件状态中的 items 数组，Form 组件接收 handleSubmit 方法作为 onSubmit 回调，用户点击+按钮就会触发该回调。

现在是时候创建子组件了，先从 List 组件入手，只要从之前的渲染方法中抽取部分代码即可。

我们以类的形式实现 List 组件，它继承自 PureComponent，因此，只有 items 数组发生变化时它才会重新渲染：

```
class List extends React.PureComponent
```

render 方法非常简单，只要遍历数组元素来生成列表即可：

```
render() {
  return (
    <ul>
      {this.props.items.map(item => <li key={item}>{item}</li>)}
    </ul>
  )
}
```

Form 组件稍微有些复杂，因为它要处理受控输入元素的状态。它也继承自 PureComponent，因此，如果父组件提供的回调函数没有改变，那么它就不会重新渲染：

```
class Form extends React.PureComponent
```

定义 constructor 方法，设置初始状态，并为受控输入元素绑定 change 事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    value: '',
  }

  this.handleChange = this.handleChange.bind(this)
}
```

change 事件处理器的实现和我们目前见到的任何受控输入元素的事件处理器类似：

```
handleChange({ target }) {
  this.setState({
    value: target.value,
  })
}
```

它接收目标元素，也就是输入元素本身，再将输入值保存到状态中。

最后，在 render 方法中声明构成表单的所有元素：

```
render() {
  return (
    <div>
      <input
        type="text"
        value={this.state.value}
        onChange={this.handleChange}
      />
```

```
        <button
          onClick={() => this.props.onSubmit(this.state.value)}
        >+</button>
      </div>
    )
  }
```

其中包含设置了值的受控输入框，以及 change 事件处理器和+按钮，该按钮会触发父组件的回调并传入当前输入值。因为这里没有其他纯粹子组件，所以我们可以直接在 render 方法中生成函数。

大功告成！如果现在运行全新创建的 Todos 组件，我们会在页面上看到和之前一样的行为，但列表和表单的状态分开了，它们只在各自的 props 改变时才会重新渲染。

举例来说，即使尝试在列表内添加数百个项，但性能不会受到影响，输入框也不会有延迟。我们只是重构了组件并稍微修改了设计思路来恰当地分离职责，就解决了性能问题。

9.4　工具与库

接下来我们将介绍一些技巧、工具和库，以便在代码库中用它们监控并提升性能。

9.4.1　不可变性

我们在前面提过，shouldComponentUpdate 配合 PureComponent 是提升 React 应用性能的最强大的工具。

唯一的问题是，PureComponent 只会浅比较 props 以及状态，也就是说，如果传入对象作为 prop，修改该对象的某个值可能不会产生我们想要的行为。

这是因为浅比较检测不到对象属性的变化，因此组件永远不会重新渲染，除非对象本身被替换。

解决这个问题的其中一种方法是使用**不可变数据**：这种数据一旦创建，就无法再修改。

举例来说，可以按以下方式设置状态：

```
const obj = this.state.obj
obj.foo = 'bar'
this.setState({ obj })
```

即使对象的 foo 属性的值发生改变，但对象引用仍然保持不变，因此浅比较检测不到这次改变。

我们可以换一种做法，每次修改对象时都创建新的实例，如下所示：

```
const obj = Object.assign({}, this.state.obj, { foo: 'bar' })
this.setState({ obj })
```

在上述示例中，我们得到了 `foo` 属性值为 `bar` 的新对象，这样浅比较就能够检测到前后的差别。

在 ES2015 和 Babel 的帮助下，用对象展开操作符可以更优雅地表达相同概念：

```
const obj = { ...this.state.obj, foo: 'bar' }
this.setState({ obj })
```

代码结构比原来的更简洁，而且效果相同；但在撰写本书时，这段代码需要转译后才能在浏览器中运行。

React 提供了一些与不可变性相关的辅助工具，大大简化了不可变对象的使用，此外，也可以使用流行库 `immutable.js`，它提供了更多强大特性，但需要学习新的 API。

9.4.2 性能监控工具

我们已经学习过如何用 React 提供的 `Perf` 插件追踪应用的性能信息。

在前面的示例中，我们用组件的生命周期钩子开启或关闭监控过程，如下所示：

```
componentWillUpdate() {
  Perf.start()
}

componentDidUpdate() {
  Perf.stop()
  Perf.printOperations()
}
```

调用关闭方法后，我们还用 `printOperations` 方法在浏览器控制台中打印了 React 当前执行了哪些 DOM 操作，以促使改动生效。

这样做效果不错，`Perf` 也确实是非常有用的工具。但为了追踪组件性能信息而占用钩子并污染代码库似乎有些多余。

最好的方案应该是不需要修改代码就能监控组件性能，为了实现这一点，可以使用 Chrome 的 `chrome-react-perf` 扩展。

在浏览器中打开以下 URL 即可安装该扩展：

https://chrome.google.com/webstore/detail/reactperf/hacmcodfllhbnekmghgdlplbdnahmhmm

这个扩展在开发者工具中新建了一个面板，在这个面板下不用编写任何代码就能简单方便地

开启或关闭 `Perf` 插件。

能够帮助我们轻松收集组件性能信息的另一个工具是 `react-perf-tool`，我们可以在应用中安装、导入并添加这个组件，它会在浏览器窗口中提供一个美观的界面，便于管理 `Perf` 插件。

该组件会在页面底部显示一个控制台，可以在这里开启或关闭监控。它不以表格的形式显示数据，而是渲染美观的图表，以便我们更容易找出哪个组件耗费了更多渲染时间。

9.4.3　Babel 插件

我们还可以安装一些有趣的 Babel 插件来提升 React 应用的性能。它们在构建时优化代码，以促进应用更快运行。

第一个插件名为 **React 常量元素转换器**，它会寻找不随 props 改变的所有静态元素，并将它们从渲染方法（或者无状态函数式组件）中抽离出来，以避免多余地调用 `createElement`。

Babel 插件的用法很直观，先用 `npm` 安装它：

```
npm install --save-dev babel-plugin-transform-react-constant-elements
```

然后编辑.babelrc 文件，添加 `plugins` 数组属性，其中包含了我们想要激活的插件列表。

本例的插件列表如下所示：

```
{
  "plugins": ["transform-react-constant-elements"]
}
```

能够帮助提升性能的第二个 Babel 插件叫作 **React 行内元素转换器**，它会将所有 JSX 声明（或者 `createElement` 调用）替换成优化过的版本，以便代码可以更快执行。

执行以下命令来安装该插件：

```
npm install --save-dev babel-plugin-transform-react-inline-elements
```

接着将它添加到.babelrc 文件的插件数组中，如下所示：

```
{
  "plugins": ["transform-react-inline-elements"]
}
```

这两个插件都只应该在生产环境中启用，因为它们会使开发环境中的调试变得很困难。

9.5　小结

关于性能的旅程已经结束了，现在我们可以优化应用，为用户提供更好的体验。

我们在本章中学习了一致性比较算法的原理，以及 React 如何总是尝试按最优路径来修改 DOM。我们还可以用 `key` 属性帮助库更好地工作。

始终牢记，用 `Perf` 插件测试应用中需要优化的部分时，要使用生产版本的 React。

找到瓶颈所在后，可以采用本章中介绍的任何一种技巧来修复问题。你能用的第一个工具就是继承 `PureComponent` 并使用不可变数据，这样组件只会在真正需要时才重新渲染。

不要忘了避免导致 `PureComponent` 低效的那些常见错误，例如，在渲染方法中生成新函数，或者将常量作为 props。

我们还学到了合理地重构和重新设计组件架构也有助于提升性能。我们的目标是创建小型组件，使它们尽可能只有单一职责。

本章最后介绍了不可变性，我们学习了用不可变数据让 `shouldComponentUpdate` 和 `shallowCompare` 发挥作用的重要性。最后，我们还介绍了一些能让应用更快运行的工具和库。

下一章将介绍测试与调试的相关内容。

第 10 章

测试与调试

10

得益于 React 的组件化开发，测试应用变得很简单。我们可以用许多工具编写 React 测试，本章将介绍其中流行的一些工具，以便你理解它们带来的好处。

Jest 是一套完备的测试框架方案，由 Facebook 的 Christopher Pojer 以及社区中的多名贡献者所维护，它致力于提供最佳的开发体验；你也可以选择用 Mocha 自行搭建框架。本章将介绍构建最佳测试环境的这两种方式。

通过学习 TestUtils 和 Enzyme，你将学习浅渲染和完整 DOM 渲染之间的区别，快照测试的原理，以及如何收集有用的代码覆盖率信息。

充分理解这些工具及其功能后，我们将为 Redux 代码库的一个组件编写测试，并探讨一些复杂场景下的常用测试方案。

阅读完本章后，你就可以从零搭建一套测试环境，并为应用的组件编写测试。

本章包含如下内容。

- 为何测试应用很重要，以及测试如何帮助开发人员更快地行动。
- 如何搭建 Jest 环境，并用 TestUtils 测试组件。
- 如何用 Mocha 搭建相同的环境。
- 什么是 Enzyme，以及为何它是测试 React 应用的必备工具。
- 如何测试真实的组件。
- Jest 快照与 Istanbul 代码覆盖率工具。
- 高阶组件和包含多层嵌套子组件的复杂页面的常用测试方案。
- React 开发者工具与一些错误处理技巧。

10.1 测试的好处

测试 Web UI 的难度一直很大。不管是单元测试还是端对端测试，由于实际情况下界面依赖

于浏览器，用户交互以及其他诸多因素导致高效的测试方案很难实现。

如果曾经试过为 Web 编写端对端测试，你就会知道想要获得一致的结果非常复杂，更不用说测试结果还常常受漏判率影响，很多因素都会导致漏判，比如网络。除此之外，改善体验、最大化用户转化率或者新增特性都需要频繁更新 UI。

如果测试难以编写和维护，开发人员就很难将测试覆盖整个应用。另一方面，测试又非常重要，因为它们能使开发人员对自己的代码开发速度和质量更有信心。如果一段代码经过充分测试（测试代码也编写得很好），那么开发人员就可以确保它能正常运行，随时可以发布。同样，测试也使代码重构更简单，因为它能确保重写代码时不会改变原有功能。

开发人员往往更注重正在开发的特性，有时很难判断应用其他部分是否会被当前改动所影响。测试有助于避免代码回退，因为它会告诉开发人员新的代码能否通过旧的测试。这样一来，开发人员在开发新特性时会更有信心，发布过程也能大大加快。

测试应用的主要功能可以使代码库更加稳固。找到新的 bug、重现并修复都不算什么难事，而覆盖测试能够确保该 bug 以后不会再出现。

好在 React（以及组件化时代）使得 UI 的测试变得更加简单、高效。测试组件和组件树不再那么费劲，因为应用的每个单独部分都有各自的职责和界限。

如果合理地开发组件，而且组件本身很纯粹，并且做到了模块化和可复用，那么就能像简单函数那样测试它们。

现代工具还为我们提供了另一项强大功能，即可以用 Node 和控制台运行测试。每个测试用例都要启动浏览器会拖慢测试过程，降低可预测性，使开发体验大打折扣；用控制台运行测试会快很多。

当在真实的浏览器中渲染组件时，只用控制台测试组件有时会产生意外行为，但根据我的经验来说，这类情况很少见。

测试 React 组件时，我们想要确保它们能正常工作，传入不同组合的 props 总能输出正确结果。

或许我们还想要覆盖组件可能拥有的各种状态。点击按钮可能会改变状态，因此我们需要编写测试来检查是否所有事件处理器都能按预期工作。 10

即使覆盖了组件的所有功能，我们还想更进一步，编写测试来验证组件在**极端情况**下的行为。举例来说，极端情况就是所有 props 都是 `null` 或者出现报错时的组件的状态。有了这类测试后，我们就能更加确信组件行为符合预期。

测试单个组件很有效，但这无法确保多个测试过的独立组件放到一起后仍然可以正常工作。我们后面将会看到，React 可以挂载整棵组件树，并测试组件的集成是否有问题。

编写测试的技巧有很多，**测试驱动开发**（test driven development，TDD）是其中十分流行的一种。采用 TDD 模式意味着先编写测试，然后再编写能够通过测试的代码。

遵循这种模式可以帮助我们写出更好的代码，因为它促使我们在实现功能前花更多精力思考设计，这往往会带来更高的质量。

10.2　用 Jest 轻松测试 JavaScript

要想学习如何正确测试 React 组件，最重要的是动手编写代码，我们将在本节中实战一番。

React 文档提到过，Facebook 的开发人员使用 Jest 来测试组件。不过，React 并没有强制你只能使用特定的测试框架，你可以随意使用自己最喜欢的测试框架。

下一节将介绍如何用 Mocha 测试组件。

为了实战使用 Jest，我们将从零搭建一个项目，安装所有依赖并编写一个组件和相关的测试。这会非常有趣！

首先，进入一个新文件夹，并执行以下命令：

```
npm init
```

生成 package.json 后就可以开始安装依赖，第一个就是 `jest` 包本身：

```
npm install --save-dev jest
```

为了让 `npm` 知道我们想用 `jest` 命令运行测试，需要在 package.json 中添加以下脚本：

```
"scripts": {
  "test": "jest"
},
```

为了能够用 ES2015 和 JSX 编写组件和测试，需要安装与 Babel 相关的所有包，这样 Jest 就能用它们转译并读懂代码。

第二批需要安装的依赖如下所示：

```
npm install --save-dev babel-jest babel-preset-es2015 babel-preset-react
```

你可能已经知道了，现在要创建.babelrc 文件，它负责告诉 Babel 项目中要使用哪些预设规则和插件。

.babelrc 内容如下所示：

```
{
  "presets": ["es2015", "react"]
}
```

现在是时候安装 React 和 ReactDOM 了，可以用它们来创建和渲染组件：

```
npm install --save react react-dom
```

搭建工作已经完成，我们可以用 ES2015 代码运行 Jest，并在 DOM 中渲染组件，不过还有一件事要做。

前面提到过，我们想用 Node 和控制台运行测试。要想实现这一点，就不能用 ReactDOM 渲染组件，因为它需要浏览器的 DOM 接口。

Facebook 团队开发了 TestUtils 工具，有了它之后，任何测试框架都可以很轻松地测试 React 组件。

我们先安装这个工具，然后看看它提供了什么功能：

```
npm i --save-dev react-addons-test-utils
```

现在测试组件所需的依赖都准备好了。TestUtils 库提供的函数可以用来浅渲染组件，或者将组件渲染到浏览器以外的独立 DOM 中。它还提供了一些工具方法，可以引用在独立 DOM 中渲染的节点，这样我们就可以根据节点的值进行断言和预测。

TestUtils 还可以模拟浏览器事件，并检查事件处理器是否正确设置。

我们开始动手创建一个组件来进行测试。

组件名为 `Button`，它会用 props 中的文本来渲染一个按钮元素，并为点击事件绑定事件处理器。我们将采取 TDD 的形式，一开始只编写结构代码，然后依据测试编写具体的实现。

因为 TestUtils 对无状态函数式组件有一些限制，所以我们要以类的形式实现这个组件。

创建 button.js 文件并导入 `React`：

```
import React from 'react'
```

定义 `Button` 组件，如下所示：

```
class Button extends React.Component
```

目前渲染方法只要返回一个 `div` 元素让组件能够运行即可：

```
render() {
  return <div />
}
```

最后，导出 `Button` 组件：

```
export default Button
```

现在可以开始测试这个组件了，我们先创建 button.spec.js 文件，并在其中编写第一条测试。

10

Jest 会在源代码文件夹中寻找以.spec、.test 结尾的文件，或者位于__tests__folder 文件夹下的文件；不过为了满足项目需要，你也可以按自己的想法修改这项配置。

在 button.spec.js 中先导入依赖：

```
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Button from './button'
```

第一项依赖是用于编写 JSX 代码的 React，第二项是 TestUtils，稍后我们将介绍如何使用它。最后一项就是刚刚创建的 `Button` 组件。

我们要编写的第一条测试用于确认一切依赖都能正常运行（我总会这样做），如下所示：

```
test('works', () => {
  expect(true).toBe(true)
})
```

`test` 函数接受两个参数，第一个参数用于描述本条测试，第二个参数就是包含实际测试代码的函数。另外，可以用 `expect` 函数对某个对象进行预测，它还可以链式调用其他方法，如 `toBe`，以检查传给 `expect` 的对象和传给 `toBe` 的对象是否相同。

打开终端，并运行以下命令：

```
npm test
```

你应该可以看到以下输出内容：

```
PASS  ./button.spec.js
  ✓ works (3ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.48s
Ran all test suites.
```

如果控制台输出 PASS，那么你可以准备编写真正的测试了。

正如之前说过，我们想要测试的是，传入一些 props 时组件能够正确渲染，并且事件处理器可以按预期工作。

测试 React 组件的方式一般有两种：

- 浅渲染；
- 将组件挂载到独立 DOM 中。

第一种方式最简单，也最好理解，我们先来学习它。浅渲染允许你按**一级深度**渲染组件，然后根据它返回的渲染结果进行一些预测。

只渲染一级深度是指将组件隔离出来测试，即使其中包含一些很复杂的子组件，它们也不会被渲染，就算它们发生变化或者加载失败，也不影响测试结果。

我们编写的第一条基础测试用于检查给定的文本是否渲染为 `button` 元素的子元素。

测试代码的开头如下所示：

```
test('renders with text', () => {
```

第一步先定义文本变量，它会作为 prop 传给组件，我们还要根据它来判断是否渲染了正确的值：

```
const text = 'text'
```

现在可以开始实现浅渲染了，它只有三行代码：

```
const renderer = TestUtils.createRenderer()
renderer.render(<Button text={text} />)
const button = renderer.getRenderOutput()
```

第一行代码创建了 `renderer` 变量，第二行传入了文本变量来渲染 `button` 组件，最后一行得到渲染结果。

渲染结果和以下对象类似：

```
{ '$$typeof': Symbol(react.element),
      type: 'button',
      key: null,
      ref: null,
      props: { onClick: undefined, children: 'text' },
      _owner: null,
      _store: {} }
```

你可能认出了这是一个 React 元素，因为它带有 `type` 属性和 prop。第二个 prop 是 `children` 元素，我们想要确保它的值是正确的。

现在我们知道了渲染结果的结构，编写预测代码就很简单了：

```
expect(button.type).toBe('button')
expect(button.props.children).toBe(text)
```

上述代码表示我们希望 `button` 组件返回一个类型属性为 `button` 的元素，并且它应该将给定文本作为子元素。

最后，关闭测试代码块：

```
})
```

如果现在切换到控制台并输入以下命令：

```
npm test
```

你应该可以看到以下内容：

```
FAIL  ./button.spec.js
 ● renders with text
   expect(received).toBe(expected)
   Expected value to be (using ===):
     "button"
   Received:
     "div"
```

测试没有通过，这和我们预想的一样，因为这是 TDD 模式第一次运行测试，此时组件还没有实现。它只返回一个 `div` 元素。接下来回到 `button` 组件中，编写代码使测试得以通过。我们对 `render` 方法进行以下修改：

```
render() {
  return (
    <button>
      {this.props.text}
    </button>
  )
}
```

然后再次运行 `npm test`，控制台上应该会出现一个绿色的勾：

```
PASS  ./button.spec.js
  ✓ renders with text (9ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.629s
Ran all test suites.
```

祝贺你！你用 TDD 模式编写的第一条组件测试通过了。

现在我们来看看如何编写测试检查按钮点击时会调用传给组件的 `onClick` 处理器。

开始编码前，我们要引入两个新的概念：mock 与独立 DOM。

前者可以简化测试过程中对函数行为的测试。这里我们要用 `onClick` 属性将一个函数传递给按钮组件，并验证用户点击按钮时是否会触发这个函数。

为了实现这一点，我们需要创建一个名为 mock 的特殊函数（又名 spy，具体取决于框架），这个函数的行为很像真正的函数，但拥有一些特殊属性。举例来说，我们可以检查它是否被调用过，以及调用的次数和参数。

Jest是一套完备的测试框架，提供了正确编写测试所需的一切工具。用Jest提供的`jest.fn()`就能创建一个 mock 函数。

编写下一步测试前要掌握的第二个概念是,我们无法用 TestUtils 在浅渲染中模拟 DOM 事件。

原因是，要想用 TestUtils 模拟事件，需要操作真正的组件，而不是简单的 React 元素。

因此，要想测试浏览器事件，需要将组件渲染到独立 DOM 中。在真正的 DOM 中渲染组件需要浏览器环境，但 Jest 提供了一个特殊的 DOM 结构，我们可以用控制台将组件渲染进去。

与浅渲染相比，将组件渲染到 DOM 中的语法稍微有些不同。我们来看看具体的测试代码。

首先，创建新的测试代码块：

```
test('fires the onClick callback', () => {
```

测试的描述表明我们将检查 `onClick` 回调能否正常工作。

然后用 `jest` 函数创建 `onClick` mock 函数：

```
const onClick = jest.fn()
```

接下来就是完全将组件渲染进 DOM：

```
const tree = TestUtils.renderIntoDocument(
  <Button onClick={onClick} />
)
```

如果在控制台打印整棵树，就会看到我们取回了真正的组件实例，而不仅仅是 React 元素。

出于同样的原因，我们不能简单地查看 `renderIntoDocument` 函数返回的对象，而要用 TestUtils 库中的函数来查找按钮元素：

```
const button = TestUtils.findRenderedDOMComponentWithTag(
  tree,
  'button'
)
```

顾名思义，这个函数会根据传入的标签名在树中查找组件。

现在我们用 TestUtils 的另一个函数来模拟事件：

```
TestUtils.Simulate.click(button)
```

`Simulate` 对象接受一个与事件同名的函数，函数的参数就是要触发事件的目标组件。

最后，编写预测代码：

```
expect(onClick).toBeCalled()
```

这里简单地表明我们希望 mock 函数被调用。

执行 `npm test` 来运行测试，但测试不会通过，因为 `onClick` 的功能还未实现。

```
FAIL  ./button.spec.js
● fires the onClick callback

   expect(jest.fn()).toBeCalled()
   Expected mock function to have been called.
```

这就是 TDD 流程的运作方式。我们回到 button.js 中并实现事件处理器，按以下方式修改 render 方法：

```
render() {
  return (
    <button onClick={this.props.onClick}>
      {this.props.text}
    </button>
  )
}
```

再次执行 npm test，此时测试通过了：

```
PASS  ./button.spec.js
  ✓ renders with text (10ms)
  ✓ fires the onClick callback (17ms)
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.401s, estimated 2s
Ran all test suites.
```

我们完整测试并正确实现了这个组件。

10.3　灵活的测试框架 Mocha

接下来我们将学习如何用 Mocha 实现相同的效果，并证明你可以用任何测试框架来测试 React。另外，本节也有利于学习两个测试框架间的主要区别，Jest 是一个高度集成的测试框架，尝试自动完成一切操作，以便开发体验更加流畅，而 Mocha 本身不关心你需要哪些工具。使用 Mocha 需要你自行决定安装哪些不同的包来正确测试 React。

创建一个新的文件夹，并用以下语句初始化一个新的 npm 包：

```
npm init
```

首先安装 mocha：

```
npm install --save-dev mocha
```

和 Jest 一样，要想使用 ES2015 和 JSX，Mocha 也需要借助 Babel。为此，先安装下列这些包：

```
npm install --save-dev babel-register babel-preset-es2015 babel-preset-react
```

安装完 Mocha 和 Babel 后，设置测试脚本，如下所示：

```
"scripts": {
  "test": "mocha --compilers js:babel-register"
},
```

我们告诉 npm 运行 mocha 测试任务，启用编译器标记设置让 Babel 记录器负责处理 JavaScript 文件。

下一步，安装 React 和 ReactDOM：

```
npm install --save react react-dom
```

再安装 TestUtils，我们要用它在测试环境中渲染组件：

```
npm install --save-dev react-addons-test-utils
```

用 Mocha 编写测试的基本功能已经准备好了，但为了用上和 Jest 一样的功能，我们还要安装三个包。

第一个是 chai，它允许我们像 Jest 那样编写预测代码。第二个是提供了 spies 功能的 chai-spies，可以用于检查 onClick 回调是否被调用过。

最后很重要的一个包是 jsdom，它可以用于创建独立 DOM，这样即便没有真实浏览器环境，也可以在控制台中使用 TestUtils：

```
npm install --save-dev chai chai-spies jsdom
```

现在可以开始编写测试了，我们会用到前面创建的 button.js。它已经完整实现了，因此这次不需要重复 TDD 流程，本节的目的在于展示两个框架间的差别。

Mocha 约定，测试用例应该放在 test 文件夹下，新建该文件夹，并放入 button.spec.js 文件。

首先，导入所有依赖：

```
import chai from 'chai'
import spies from 'chai-spies'
import { jsdom } from 'jsdom'
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import Button from '../button'
```

你可能已经注意到了，导入项比 Jest 多了不少，因为 Mocha 允许你自由选择想要的工具。

下一步，让 chai 使用 spy：

```
chai.use(spies)
```

接着从 chai 中引用我们需要的功能。实现测试用例时会用到它们：

下一步，设置 `jsdom`，并提供渲染 React 组件所需的 DOM 对象：

```
global.document = jsdom('')
global.window = document.defaultView
```

一切就绪后就可以开始编写第一条测试了。Mocha 最典型的地方就是要用两个函数定义测试：第一个函数是 `describe`，它负责封装相同模块的一系列测试用例；另一个函数是 `it`，它包含具体的测试代码。

在本例中，我们先描述按钮的行为：

```
describe('Button', () => {
```

然后编写第一条测试，这里我们希望元素类型和文本是正确的：

```
it('renders with text', () => {
```

定义文本变量，以判断预测代码是否有效：

```
const text = 'text'
```

接着像之前一样浅渲染组件：

```
const renderer = TestUtils.createRenderer()
renderer.render(<Button text={text} />)
const button = renderer.getRenderOutput()
```

最后，完成预测代码：

```
expect(button.type).to.equal('button')
expect(button.props.children).to.equal(text)
```

如你所见，这里的语法有些不同。我们没有用 `toBe` 方法，而是链式调用了 `chai` 提供的 `to.equal` 方法。但两者效果一样，都是比较两个值是否相等。

现在可以闭合第一个测试代码块了：

```
})
```

第二条测试负责测试 `onClick` 回调是否被触发，如下所示：

```
it('fires the onClick callback', () => {
```

接着创建 spy，做法和之前一样：

```
const onClick = spy()
```

用 TestUtils 将按钮渲染进独立 DOM：

```
const tree = TestUtils.renderIntoDocument(
  <Button onClick={onClick} />
)
```

再根据标签名在 tree 对象中查找组件：

```
const button = TestUtils.findRenderedDOMComponentWithTag(
  tree,
  'button'
)
```

下一步，模拟按钮点击：

```
TestUtils.Simulate.click(button)
```

最后，完成预测代码：

```
expect(onClick).to.be.called()
```

这里的语法还是有些不同，但概念依然保持不变：检查是否调用过 spy 函数。

现在用 Mocha 运行 root 文件夹下的 npm test 命令，我们应该可以看到以下信息：

```
Button
  ✓ renders with text
  ✓ fires the onClick callback

2 passing (847ms)
```

这表明我们的测试已经通过，现在可以开始用 Mocha 测试真正的组件了。

10.4 React JavaScript 测试工具

现在，你应该很清楚如何用 Jest 和 Mocha 测试基础组件，以及两者分别有何优缺点。

你也学习了 TestUtils 库的使用，以及浅渲染与完整 DOM 渲染的区别。

你可能还发现，虽然对测试组件来说很有用，但 TestUtils 用起来比较繁琐，而且有时很难正确获取元素及其属性的引用。

由于这些原因，AirBnb 的开发人员开发了 Enzyme，它基于 TestUtils 构建，可以更方便地操作渲染后的组件。

它的 API 设计得和 jQuery 一样棒，并提供了许多实用工具来操作组件，以及其状态和属性。

我们来看看如何将 Jest 测试从 TestUtils 切换到 Enzyme。

现在回到之前创建的 Jest 项目文件夹中，安装 enzyme：

```
npm install --save-dev enzyme
```

打开 button.spec.js，将导入语句修改为如下所示的代码：

```
import React from 'react'
import { shallow } from 'enzyme'
import Button from './button'
```

如你所见，这里不再导入 TestUtils，而是从 Enzyme 导入 `shallow` 函数。顾名思义，`shallow` 函数就是提供浅渲染功能的函数，此外它还有一些特殊的特性。

首先，Enzyme 可以在浅渲染元素中模拟事件，TestUtils 无法实现这一点。最重要的是，该函数返回的对象名为 ShallowWrapper，而不是简单的 React 元素。这个特殊对象提供了一些有用的属性和函数，接下来我们将介绍它们。

我们开始修改测试代码，先从**渲染文本**的用例入手。第一行代码保持不变，因为我们仍然需要文本变量：

```
const text = 'text'
```

浅渲染部分更加简单直观。原先用 TestUtils 需要三行代码，现在只要一行就够了：

```
const button = shallow(<Button text={text} />)
```

按钮就是一个 ShallowWrapper 对象，接下来我们会在新的预测语句中用到其方法：

```
expect(button.type()).toBe('button')
expect(button.text()).toBe(text)
```

如你所见，这里没有直接检查对象属性（随着 React 的更新，属性可能会发生改变），而是调用了抽象相应功能的工具函数。

`type` 函数返回渲染元素的类型，`text` 函数返回组件内渲染的文本。在本例中，它就是作为 prop 的文本变量。

完整的测试如下所示：

```
test('renders with text', () => {
  const text = 'text'
  const button = shallow(<Button text={text} />)

  expect(button.type()).toBe('button')
  expect(button.text()).toBe(text)
})
```

这比之前简洁不少。

下一步要更新 `onClick` 事件处理器的测试。第一行代码还是保持不变：

```
const onClick = jest.fn()
```

这里仍然要用 Jest 创建 mock 函数，以便在预测代码中使用 spy 功能。

原先用 `renderIntoDocument` 在独立 DOM 中渲染组件，现在替换为以下方法：

```
const button = shallow(<Button onClick={onClick} />)
```

现在也不需要用 `findRenderedDOMComponentWithTag` 查找按钮组件，因为浅渲染已经引用它了。

模拟事件的语法和 TestUtils 有些不同，但仍然很直观：

```
button.simulate('click')
```

每个 ShallowWrapper 对象都有一个 `simulate` 方法供我们调用，可以传入事件名，也可以用其他数据作为第二参数。本例不需要任何第二参数，不过，我们在后文测试表单时会用到它。

最后，预测语句保持原样：

```
expect(onClick).toBeCalled()
```

完成后的代码如下所示：

```
test('fires the onClick callback', () => {
  const onClick = jest.fn()
  const button = shallow(<Button onClick={onClick} />)

  button.simulate('click')
  expect(onClick).toBeCalled()
})
```

迁移到 Enzyme 的过程相对来说比较简单，同时也提高了代码的可读性。

该库提供了一些非常有用的 API，可以查找嵌套元素或者用选择器根据类名和类型查找元素。

还有一些方法可以对 props 进行断言和预测，并为组件注入任意状态和 context 对象。

浅渲染可以满足大部分使用场景，除了它以外，库还提供了一个 `mount` 方法，用于在 DOM 中渲染组件树。

10.5　真实测试示例

我们编写了可用的测试，并清晰地理解了各种工具和库的用途。接着我们来测试真实的组件。

上文用到的 `Button` 组件实例值得参考，我们应该始终尽量保持组件简洁；不过有时也需要实现各种逻辑和状态，这就给测试带来了挑战。

我们将要测试的组件是 Redux TodoMVC 示例中的 TodoTextInput：

https://github.com/reactjs/redux/blob/master/examples/todomvc/src/components/TodoTextInput.js

直接将它复制到 Jest 项目文件夹下。

这个示例非常经典，因为它带有多种 props，类名会随着接收的 props 变化，并且还有三个事件处理器，我们可以测试这三个事件处理器所实现的逻辑。

TodoMVC 示例代表了创建真实应用的**标准形式**，它可以用不同的框架实现，从而比较它们的特性，且更方便开发人员进行选择。

该示例最终完成的是一个很简单的应用，用户可以添加待办事项或标记已办事项。我们要关注的组件是其中用于添加和编辑事项的文本输入框。

开始测试前应该先快速介绍一下该组件的实现，以便我们能够理解其功能。

首先，用类来定义该组件：

```
class TodoTextInput extends Component
```

`propTypes` 定义为类的静态属性：

```
static propTypes = {
  onSave: PropTypes.func.isRequired,
  text: PropTypes.string,
  placeholder: PropTypes.string,
  editing: PropTypes.bool,
  newTodo: PropTypes.bool
}
```

要想让 Babel 识别类属性，需要使用 `transform-class-properties` 插件，可以直接安装它：

```
npm install --save-dev babel-plugin-transform-class-properties
```

然后将它添加到.babelrc 中的 Babel 插件列表中：

```
"plugins": ["transform-class-properties"]
```

状态同样定义为一个 `class` 属性：

```
state = {
  text: this.props.text || ''
}
```

其默认值为 props 的文本属性或空字符串。

然后是三个事件处理器，它们同样定义为 `class` 属性，并且用箭头函数来实现，这样就不需要在 `constructor` 方法中手动绑定 `this` 了。

第一个是 `submit` 事件处理器：

```
handleSubmit = e => {
  const text = e.target.value.trim()
```

```
    if (e.which === 13) {
      this.props.onSave(text)
      if (this.props.newTodo) {
        this.setState({ text: '' })
      }
    }
  }
```

处理器函数接收事件对象。它会去除目标元素值中多余的空白字符，然后检查触发当前事件的按键是否为**回车键**（键码 13），如果满足该条件，就将处理过的值传给 props 上的 `onSave` 函数。如果 prop `newTodo` 为 `true`，则将状态重新设成空字符串。

第二个是 `change` 事件处理器：

```
handleChange = e => {
  this.setState({ text: e.target.value })
}
```

除了定义为 `class` 属性，你应该很熟悉这个方法了，它负责更新受控输入元素的状态。

最后是 `blur` 事件处理器：

```
handleBlur = e => {
  if (!this.props.newTodo) {
    this.props.onSave(e.target.value)
  }
}
```

当 props 的 `newTodo` 属性为 `false` 时，它会用输入框的值触发 `onSave` 函数。

现在定义 `render` 方法，输入元素上设置了它的所有属性：

```
render() {
  return (
    <input className={
      classnames({
        edit: this.props.editing,
        'new-todo': this.props.newTodo
      })}
      type="text"
      placeholder={this.props.placeholder}
      autoFocus="true"
      value={this.state.text}
      onBlur={this.handleBlur}
      onChange={this.handleChange}
      onKeyDown={this.handleSubmit} />
  )
}
```

这里用 `classnames` 函数设置类名。它来自 Jed Watson 开发的一个非常有用的库，可以很方便地通过条件逻辑设置类名。

接着设置 type 和 autofocus 这类静态属性，并用状态中的 text 属性控制输入元素的值，每次改变都会更新该属性。最后，为元素的事件属性绑定相应的事件监听器。

开始前需要注意的是，我们要搞清楚测试的目的和原因。查看这个组件，不难发现哪些部分需要重点测试。拿本例来说，可以将它看作从其他团队接手的或者加入新公司时发现的遗留代码。

以下列表或多或少地列出了该组件值得测试的所有状态和功能：

- 用 props 的值初始化状态；
- 元素正确使用了 placeholder 属性；
- 类名和条件逻辑相匹配；
- 输入框的值改变时，状态会随之更新；
- onSave 回调会由不同的状态和条件所触发。

现在我们开始编写代码，创建 TodoTextInput.spec.js 文件并写入以下语句：

```
import React from 'react'
import { shallow } from 'enzyme'
import TodoTextInput from './TodoTextInput'
```

这里导入了 React、Enzyme 的浅渲染函数，以及待测组件。另外我们还创建了一个工具函数，在某些测试中将它传递给必要属性 onSave：

```
const noop = () => {}
```

现在开始编写第一条测试，以检查元素的默认值是否为文本属性的值：

```
test('sets the text prop as value', () => {
  const text = 'text'
  const wrapper = shallow(
    <TodoTextInput text={text} onSave={noop} />
  )

  expect(wrapper.prop('value')).toBe(text)
})
```

代码非常直观：创建文本变量，将它传递给浅渲染的组件。如你所见，这里为必要属性 onSave 传递了工具函数 noop，虽然它没有实际用处，但如果传入 null，React 会给出警告。

最后，渲染组件并检查输出元素的值属性是否与给定文本一致。如果此时在控制台中运行 npm test 命令，那么就能看到测试通过的提示：

```
PASS  ./TodoTextInput.spec.js
  ✓ sets the text prop as value (10ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.384s
Ran all test suites.
```

太棒了，我们接着编写其他测试。第二条和第一条类似，只不过这次要测试 placeholder 属性：

```
test('uses the placeholder prop', () => {
  const placeholder = 'placeholder'
  const wrapper = shallow(
    <TodoTextInput placeholder={placeholder} onSave={noop} />
  )

  expect(wrapper.prop('placeholder')).toBe(placeholder)
})
```

再次运行 npm test，控制台会提示两条测试都通过了。

现在我们来研究一些更有趣的内容，看看传入某些 props 后，元素是否会新增相应类名：

```
test('applies the right class names', () => {
  const wrapper = shallow(
    <TodoTextInput editing newTodo onSave={noop} />
  )

  expect(wrapper.hasClass('edit new-todo')).toBe(true)
})
```

本次测试为组件添加了 editing 和 newTodo 这两个属性，然后在 expect 函数内检查输出元素是否新增了相应类名。

也可以单独检查每个类，以免它们相互影响，你应该理解这点。

接下来的测试更复杂，因为我们要测试按下按键时组件的行为。

具体来说，我们要测试按下 Enter 键时，是否会用元素的值调用 onSave 回调。

```
test('fires onSave on enter', () => {
  const onSave = jest.fn()
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('keydown', { target: { value }, which: 13 })

  expect(onSave).toHaveBeenCalledWith(value)
})
```

上述代码先调用 jest.fn() 创建了一个 mock 函数。接着用值变量保存元素的值，我们还要判断调用函数的参数值是否和它一致。然后渲染组件，并传入事件对象来模拟按键事件。

事件对象拥有两个属性：带有一个 value 属性的 target 和 which，前者表示发生该事件的元素，后者表示按键的键码。

这里需要预测的是，mock 回调函数 onSave 用事件值进行了调用。

运行 npm test 会看到目前的四条测试都可以通过。

还可以编写一条类似的测试，如果按键键码不是 Enter，则什么也不会发生：

```
  test('does not fire onSave on key down', () => {
  const onSave = jest.fn()
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('keydown', { target: { value: '' } })

  expect(onSave).not.toBeCalled()
})
```

这条测试和上一条非常相似，但需要留意一下预测语句。这里用到了一个新属性.not，它对跟在自己后方的函数进行反向断言；以这里的 toBeCalled 为例，其结果应该是 false。

如你所见，编写预测代码的方式很接近口语。

五条测试通过后，就可以开始编写下一条：

```
test('clears the value after save if new', () => {
  const value = 'value'
  const wrapper = shallow(<TodoTextInput newTodo onSave={noop} />)

  wrapper.simulate('keydown', { target: { value }, which: 13 })

  expect(wrapper.prop('value')).toBe('')
})
```

与前一条测试不同的是，现在元素上多了 newTodo 属性，它会导致元素值重置。

在控制台中运行 npm test，我们会看到六条测试都通过的提示。

以下测试很常见：

```
test('updates the text on change', () => {
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={noop} />)

  wrapper.simulate('change', { target: { value } })

  expect(wrapper.prop('value')).toBe(value)
})
```

它检查受控输入元素是否正常工作，如果回想一下第 6 章中曾讨论过的内容，你就知道应用内的所有表单都要有这样的测试。

模拟 change 事件，并传入预设的值变量，然后检查输出元素的 value 属性是否与它相等。

现在已经有七条测试可以通过了，还需要完成一条。

最后一条测试用于检查，当没有新待办事项时，才会触发 blur 事件的回调：

```
test('fires onSave on blur if not new', () => {
  const onSave = jest.fn()
  const value = 'value'
  const wrapper = shallow(<TodoTextInput onSave={onSave} />)

  wrapper.simulate('blur', { target: { value } })

  expect(onSave).toHaveBeenCalledWith(value)
})
```

创建 mock 函数和期望值，用 target 属性模拟 blur 事件，然后检查是否用给定值调用过 onSave 回调。

现在运行 npm test，所有测试应该都能通过，这份列表已经很长了：

```
PASS  ./TodoTextInput.spec.js
  ✓ sets the text prop as value (10ms)
  ✓ uses the placeholder prop (1ms)
  ✓ applies the right class names (1ms)
  ✓ fires onSave on enter (3ms)
  ✓ does not fire onSave on key down (1ms)
  ✓ clears the value after save if new (5ms)
  ✓ updates the text on change (1ms)
  ✓ fires onSave on blur if not new (2ms)
Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        2.271s
Ran all test suites.
```

非常好，我们用测试覆盖了组件。如果需要重构、修改或者新增特性，这些测试会帮助我们发现新代码是否会破坏任何旧的功能。

这使我们对自己的代码更有信心，不必担心失误会导致代码回退，可以放心地修改任何一行代码了。

10.6 React 组件树快照测试

我们已经见识过真实的测试示例，但你可能会觉得花费大量时间为单个组件编写这么多测试并不值得。

检查文本、值还有类名的各种状态十分耗时费力，覆盖所有实例还需要大量代码。然而大多数时候，测试组件的最重要作用就是输出结果要正确，并且不会意外改变。Jest 引入了一项名为**快照测试**的新特性来解决这个问题。

快照就是特定时间传入某些 props 后的组件的**照片**。每次运行测试时，Jest 会创建新的**照片**，并与上一次的进行比较，以检查是否有所变化。

快照内容由 `react-test-renderer` 包的 `render` 方法输出，可以用以下命令来安装这个包：

```
npm install --save-dev react-test-renderer
```

安装完毕后，新建文件 TodoTextInput-snapshot.spec.js，写入下列导入语句：

```
import React from 'react'
import renderer from 'react-test-renderer'
import TodoTextInput from './TodoTextInput'
```

我们导入 React 来使用 JSX 语法，`renderer` 负责创建生成快照所需的组件树，最后一个是待测的目标组件。

现在所有依赖准备就绪，我们开始编写一个简单的测试：

```
test('snapshots are awesome', () => {
```

第一行代码用前面导入的 `renderer` 渲染组件：

```
const component = renderer.create(
  <TodoTextInput onSave={() => {}} />
)
```

上述代码会返回组件的实例，该组件有一个特殊方法 `toJSON`，我们接着调用它：

```
const tree = component.toJSON()
```

`tree` 变量保存原有组件返回的 React 元素，Jest 会用它生成快照，以便后续进行比较。

如果将 `tree` 变量输出到控制台中，它的样子如下所示：

```
{ type: 'input',
        props:
         { className: '',
           type: 'text',
           placeholder: undefined,
           autoFocus: 'true',
           value: '',
           onBlur: [Function],
           onChange: [Function],
           onKeyDown: [Function] },
        children: null }
```

最后，编写预测语句检查 `tree` 变量的内容是否匹配先前保存的快照：

```
expect(tree).toMatchSnapshot()
```

第一次用 `npm test` 命令运行测试时，快照是全新创建的，保存在_snapshots_文件夹下。

该文件夹下的每个文件都表示一张快照。快照文件中保存的渲染输出结果不是 React 元素对象，而是可读性更好的版本：

```
exports[`test snapshots are awesome 1`] = `
<input
  autoFocus="true"
  className=""
  onBlur={[Function]}
  onChange={[Function]}
  onKeyDown={[Function]}
  placeholder={undefined}
  type="text"
  value="" />
`;
```

回到测试，为组件添加 `editing` 属性，并再次执行 `npm test` 命令，控制台会给出以下响应：

```
FAIL  ./TodoTextInput-snapshot.spec.js
 ● snapshots are awesome
   expect(value).toMatchSnapshot()
   Received value does not match stored snapshot 1.
   - Snapshot
   + Received
   @@ -1,8 +1,8 @@
    <input
      autoFocus="true"
   -  className=""
   +  className="edit"
      onBlur={[Function]}
      onChange={[Function]}
      onKeyDown={[Function]}
      placeholder={undefined}
      type="text"
```

它向我们展示了本次快照有什么变化。具体来说，就是 `className` 属性之前为空，而现在多了字符串 `edit`。

再往下我们还会看到这条信息：

```
Inspect your code changes or run with npm test -- -u to update them.
(审查代码中的改动，或者执行 npm test -- -u 来更新代码。)
```

快照使开发者体验变得极其流畅；只要一个简单的标记，我们就能确认新的快照是否对应组件的正确版本。执行 `npm test -- -u` 可以自动更新快照。

如果组件被错误地修改了，我们可以回到代码中修复它。

如你所见，快照测试是一项很强大的特性，能够简化组件测试，使开发人员无须编写测试覆盖所有组件状态，大大节约了时间。

10.7　代码覆盖率工具

编写测试的原因很多，我们已经在前面介绍了一部分。其中最主要的原因就是能始终为代码库贡献价值，使其更稳健。

因为这一点，我始终对统计测试用例数量、代码行数以及测试覆盖率持怀疑态度。我建议大家不要将重点放在数量上，而应该注重测试能提供什么价值。

但在有些场景下，获取覆盖率指标以及追踪测试用例数量还是有用的。在包含许多不同模块的大型项目中，这样做更便于定位没有经过充分测试，甚至从未测试过的那些文件。

再次强调，Jest 为你提供了运行测试所需的一切工具，当然，它也提供了测量并保存代码覆盖率信息的功能。

它用到了 Istanbul，这是最流行的代码覆盖率库之一，如果使用的是 Mocha，那么你需要自己手动安装这个库。

运行 Jest 的代码覆盖率功能非常简单：只需要在 `npm` 脚本的 `Jest` 命令后面加上`- coverage` 标记即可。你也可以在 package.json 中为 Jest 创建配置，并将 `collectCoverage` 选项设为 `true`：

```
"jest": {
  "collectCoverage": true
}
```

再次执行 `npm test`，现在你会在控制台中看到不一样的输出，即一张显示覆盖率信息的表格：

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	100	87.5	100	100	
TodoTextInput.js	100	87.5	100	100	

如你所见，我们的代码文件几乎被完整覆盖了。第一列显示覆盖了多少条语句；第二列显示了条件语句的不同分支的覆盖率；第三列表示测试过多少函数；第四列显示了测试覆盖的代码行数。最后一列目前为空，用于表示测试没有覆盖的代码行数，当我们想快速找到代码中需要更多关注的部分时，这列信息就非常有用了。

目前唯一没有达到 100%覆盖率的只有分支一列，实际上，我在最后一条测试中故意没有覆盖一个分支，以便我们可以一起来实现完整的覆盖率。

打开 TodoTextInput.js 文件，并查看 `onBlur` 处理器，你会注意到它包含两个分支：

```
handleBlur = e => {
  if (!this.props.newTodo) {
```

```
      this.props.onSave(e.target.value)
    }
  }
```

如果待办事项不是新增的，onSave 函数会被输入框的当前值调用；而如果待办事项是新增的，则什么也不会执行。

我们只测试了前一条最为明显的路径，但往往测试所有不同路径是很有价值的，这样可以确保组件在一切情况下都能正常工作。

回到 TodoTextInput.spec.js 中，并新增一条测试：

```
test('does not fire onSave on blur if new', () => {
  const onSave = jest.fn()
  const wrapper = shallow(
    <TodoTextInput newTodo onSave={onSave} />
  )

  wrapper.simulate('blur')

  expect(onSave).not.toBeCalled()
})
```

这条测试和文件中的上一条很相似，只不过这里是向组件传入 newTodo 属性，然后再检查 onSave 回调是否被调用。

如果再次执行 npm test，我们会看到表格的每列都达到了 100%。

10.8　常用测试方案

接下来是有关测试的最后一节内容，我们将了解测试一些复杂组件时应该掌握的常用模式。

现在你应该很熟悉组件测试了，也应该掌握了为应用编写测试的所有知识。然而，有时很难找到最佳测试策略，如高阶组件。

10.8.1　测试高阶组件

前文提过，我们可以用高阶组件在应用的不同组件间共享功能。高阶组件就是函数，它们接收组件并返回增强后的版本。

因为这类组件测试起来没有简单组件那么直观，所以值得我们一起学习一些常用方案。

接下来要测试的目标组件是第 5 章中创建的高阶组件 withData。我们会对它获取数据的方式进行一些小小的改动。

withData 函数具有以下特点：

```
const withData = URL => Component => (...)
```

该函数接收数据加载路径的 URL，并将这项数据传给目标组件。URL 参数可以是接收当前 props 对象的函数，也可以是静态字符串。

withData 函数返回如下定义的类：

```
class extends React.Component
```

在 constructor 方法中初始化数据：

```
constructor(props) {
  super(props)

  this.state = { data: [] }
}
```

生命周期钩子 componentDidMount 负责加载数据：

```
componentDidMount() {
  const endpoint = typeof url === 'function'
    ? url(this.props)
    : url

  getJSON(endpoint).then(data => this.setState({ data }))
}
```

如你所见，这里和第 5 章中的示例稍有不同，我们没有直接使用获取函数，而是调用了 getJSON 函数，这样便于你学习如何模拟外部模块。

另外，最佳实践要求将第三方库和 API 调用封装或抽象到独立模块中，以便在测试时隔离组件及其依赖。

在文件顶部导入 getJSON 函数：

```
import getJSON from './get-json'
```

它返回 promise 对象，其中包含了请求路径返回的 JSON 数据。

最后，展开 props 和状态对象，调用 render 渲染目标组件：

```
render() {
  return <Component {...this.props} {...this.state} />
}
```

现在，我们想要用测试覆盖这个函数的地方和之前有些不同，先从最简单的开始。举个简单的例子，检查增强后的组件接收到的 props 是否正确传递给了目标组件。

接下来测试根据 URL 生成请求路径的逻辑，看看它是否适用于函数和静态字符串这两种情况。

最后我们要测试的是，一旦 getJSON 函数返回数据，目标组件就能接收到它。

先在测试文件中加载所有依赖：

```
import React from 'react'
import { shallow, mount } from 'enzyme'
import withData from './with-data'
import getJSON from './get-json'
```

这里同时导入了 Enzyme 的 shallow 函数和 mount 函数,因为这几个简单测试不需要 DOM 就能运行，另外要用 mount 函数测试生命周期钩子做了什么。

接着创建测试过程要用到的一些变量：

```
const data = 'data'
const List = () => <div />
```

data 变量就是模拟数据，用于检查获取到的数据是否正确地传给了组件，此外还有一个空的 List 组件。

测试高阶组件时，创建空组件是很常见的做法，因为我们要增强一个目标组件，这样才能判断所有特性是否都能正常运行。

接下来是最难，也最强大的部分：

```
jest.mock('./get-json', () => (
  jest.fn(() => ({ then: callback => callback(data) }))
))
```

之前提过要用外部模块来获取数据。我们不希望加载真实数据，最重要的是，我们不希望外部模块因为某些原因不可用，进而导致测试失败。用 Jest 就可以很方便地隔离并模拟依赖。

调用 jest.mock 函数，让测试框架将外部模块替换成作为第二个参数的函数。该函数返回 jest.fn 创建的 mock 函数，mock 函数会返回类似 promise 的对象，只不过这个对象是同步的。它拥有 then 函数，后者可以触发接收的回调并传入前面定义的伪数据。

从现在起，我们就可以对高阶组件进行单元测试，无须担心 getJSON 函数的行为或 bug。

现在我们准备开始编写真正的测试，第一条将检查 props 是否正确地传给了目标组件：

```
test('passes the props to the component', () => {
  const ListWithGists = withData()(List)
  const username = 'gaearon'

  const wrapper = shallow(<ListWithGists username={username} />)

  expect(wrapper.prop('username')).toBe(username)
})
```

代码清晰易懂，不过我们还是一起来详细解读一下。首先，我们将空组件 `List` 传给了高阶组件并增强，然后定义一个 prop 传给组件，接下来进行浅渲染。

最后，检查输出结果是否有值相同的 prop。非常好，此时执行 `npm test` 命令会看到第一条测试通过。

接下来的测试需要将组件挂载进独立 DOM。我们先检查函数和静态字符串形式的 URL 参数是否都可用。静态字符串测试起来很简单：

```
test('uses the string url', () => {
  const url = 'https://api.github.com/users/gaearon/gists'
  const withGists = withData(url)
  const ListWithGists = withGists(List)

  mount(<ListWithGists />)

  expect(getJSON).toHaveBeenCalledWith(url)
})
```

定义 `url` 变量，并用偏函数写法生成新函数；接着用这个函数增强 `List` 组件。

然后挂载组件，并检查是否用传入的 URL 调用了 `getJSON` 函数。很简单：两条测试都通过了。

现在我们来检查 URL 函数是否可用：

```
test('uses the function url', () => {
  const url = jest.fn(props => (
    `https://api.github.com/users/${props.username}/gists`
  )
  const withGists = withData(url)
  const ListWithGists = withGists(List)
  const props = { username: 'gaearon' }

  mount(<ListWithGists {...props} />)

  expect(url).toHaveBeenCalledWith(props)
  expect(getJSON).toHaveBeenCalledWith(
    'https://api.github.com/users/gaearon/gists'
  )
})
```

先用 Jest 的 mock 功能生成 URL 函数，以便为它编写预测，然后增强 `List` 组件并定义传给它的 props。

结尾是两条预测语句：

- 第一条检查是否用给定的 props 调用了 URL 函数；
- 第二条再次检查是否用正确的请求路径触发了 `getJSON` 函数。

最后一条测试用于检查返回给 getJSON 模块的数据是否正确地传给了目标组件：

```
test('passes the data to the component', () => {
  const ListWithGists = withData()(List)

  const wrapper = mount(<ListWithGists />)

  expect(wrapper.prop('data')).toEqual(data)
})
```

上述代码先用高阶组件增强了 List 组件，然后挂载组件并保存封装器的引用。接着在挂载的封装器中查找 List 组件，并检查它的 data 属性是否与请求获取的数据一致。

再次执行 npm test 命令，我们会看到四条测试都通过了。

10.8.2 页面对象模式

现在我们来看看，当组件树变得更复杂，而且有多层嵌套的子组件时，有哪些常见的测试编写方式。

接下来的示例要用到第 6 章中创建的**受控表单组件**：

```
class Controlled extends React.Component
```

我们先来快速浏览一遍它的功能来回顾其工作原理，然后再谈测试。

constructor 方法初始化了状态并绑定了事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    firstName: 'Dan',
    lastName: 'Abramov',
  }

  this.handleChange = this.handleChange.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
}
```

handleChange 处理器负责更新状态中的输入框的值：

```
handleChange({ target }) {
  this.setState({
    [target.name]: target.value,
  })
}
```

此外还有 handleSubmit 处理器，它会调用事件对象的 preventDefault 函数来禁用提交

表单时的浏览器的默认行为。然后调用父组件传来的 onSubmit 函数，并传入拼接过的输入值。

原示例中没有 onSubmit 这个函数，不过现在我们要用它展示如何正确测试组件：

```
handleSubmit(e) {
  e.preventDefault()

  this.props.onSubmit(
    `${this.state.firstName} ${this.state.lastName}`
  )
}
```

最后，用 render 方法定义输入框并绑定处理器函数：

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <input
        type="text"
        name="firstName"
        value={this.state.firstName}
        onChange={this.handleChange}
      />
      <input
        type="text"
        name="lastName"
        value={this.state.lastName}
        onChange={this.handleChange}
      />
      <button>Submit</button>
    </form>
  )
}
```

这里要测试的一项基本功能是，在输入框中输入内容并提交表单将会用输入值触发 onSubmit 回调。

现在你应该很清楚如何用 Enzyme 编写测试来覆盖这种场景，我们来一起看看：

```
test('submits the form', () => {
```

一开始先用 Jest 定义模拟的 onSubmit 函数，挂载组件并保存封装器的引用：

```
const onSubmit = jest.fn()
const wrapper = shallow(<Controlled onSubmit={onSubmit} />)
```

然后找到第一个输入框，触发它的 change 事件，并传入我们想要更新的值：

```
const firstName = wrapper.find('[name="firstName"]')
firstName.simulate(
  'change',
  { target: { name: 'firstName', value: 'Christopher' } }
)
```

第二个输入框同理：

```
const lastName = wrapper.find('[name="lastName"]')
lastName.simulate(
  'change',
  { target: { name: 'lastName', value: 'Chedeau' } }
)
```

输入框都更新完毕后，模拟提交表单事件：

```
const form = wrapper.find('form')
form.simulate('submit', { preventDefault: () => {} })
```

现在我们来编写预测语句：

```
expect(onSubmit).toHaveBeenCalledWith('Christopher Chedeau')
```

记得闭合测试代码块：

```
})
```

在控制台执行 `npm test` 会显示测试通过的提示，这正是我们所期望的；但如果再看看测试的实现，你就能轻易发现其中的一些问题和潜在的优化点。

最明显的地方就是填充输入框的代码重复了，只是有些变量不同。这种代码太繁琐，更重要的是，它和标记结构耦合了。

如果这种测试很多，那么只要标记发生改变，就要修改文件许多部分的代码。如果去除重复，并将选择器移到同一个地方，表单改变时修改选择器会更方便，这样做不是更好吗？

这里**页面对象模式**就能派上用场了。如果我们创建一个页面对象来表示页面元素，并隐藏选择器，再用它填充表单并提交，这样做有很多好处，也避免了代码重复。

客观来说，在测试中应用 **DRY 原则**往往不是最好的做法，因为这可能会引发更多 bug 并提升复杂度，不过就本例而言，这么做是很有价值的。

我们来看看页面对象模式如何改进受控表单的测试。

首先，用类创建一个 `Page` 对象：

```
class Page
```

类的 `constructor` 方法从 Enzyme 接收顶层的 `wrapper` 变量并保存，以便接下来可以使用：

```
constructor(wrapper) {
  this.wrapper = wrapper
}
```

10

接着定义一个通用函数来填充输入框，该函数接受 name 和 value 参数，然后触发 change 事件：

```
fill(name, value) {
  const field = this.wrapper.find(`[name="${name}"]`)
  field.simulate('change', { target: { name, value } })
}
```

实现 submit 函数，以便对查找按钮以及模拟浏览器事件的部分进行抽象：

```
submit() {
  const form = this.wrapper.find('form')
  form.simulate('submit', { preventDefault() {} })
}
```

现在可以按以下方式重写原来的测试了：

```
test('submits the form with the page object', () => {
  const onSubmit = jest.fn()
  const wrapper = shallow(<Controlled onSubmit={onSubmit} />)

  const page = new Page(wrapper)
  page.fill('firstName', 'Christopher')
  page.fill('lastName', 'Chedeau')
  page.submit()

  expect(onSubmit).toHaveBeenCalledWith('Christopher Chedeau')
})
```

如你所见，这里创建了页面对象的实例，我们用它的函数填充输入框并提交表单。

页面对象使代码变得更简洁，没有多余的重复。如果组件发生了改变，我们不必修改多个测试，只要简单直观地修改页面对象的工作方式即可。

10.9　React 开发者工具

只在控制台中进行测试还不够，我们想要检查应用在浏览器中的运行情况，此时可以使用 React 开发者工具。

这个工具是一个 Chrome 扩展，可以从以下 URL 安装：

https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en

安装完成后，Chrome 开发者工具会多出一个 React 标签页，你可以在这里审查渲染的组件树，也可以检查组件在特定时间的内部状态与接收的属性。

可读取 props 和状态对象，也可以实时修改它们以触发 UI 更新并直接查看结果。

这是一个必备工具，最新的版本还新增了一项特性，勾选 Trace React Updates 选框即可启用。

启用这项功能后，就可以在使用应用时直观地看到执行某个特殊操作会更新哪个组件。更新后的组件会用红色矩形框高亮显示，这样就能很方便地定位需要优化的地方。

10.10 React 错误处理

即便代码写得很完美，也覆盖了完整的测试，但错误仍然会出现。不同的浏览器和环境，以及真实用户数据都是我们无法控制的因素，它们有时会导致代码运行出错。作为开发人员，我们必须接受这一点。

应用出现问题时，最好的解决办法是：

- 通知用户，帮助他们理解发生了什么情况，并告诉他们应该怎么做；
- 收集与错误相关的一切信息以及应用状态，以便快速重现并修复 bug。

React 的错误处理方式一开始会让人觉得有些违反直觉。

假设我们有以下组件：

```
const Nice => <div>Nice</div>
```

以及：

```
const Evil => (
  <div>
    Evil
    {this.does.not.exist}
  </div>
)
```

将以下 App 组件渲染进 DOM，我们看看会发生什么事情：

```
const App = () => (
  <div>
    <Nice />
    <Evil />
    <Nice />
  </div>
)
```

举例来说，我们可能认为第一个 Nice 组件能成功渲染，接着 Evil 组件抛出异常，这会导致渲染停止。或者说两个 Nice 组件都能渲染，而 Evil 组件不会。但实际情况是屏幕上什么都不会显示。

在 React 中，如果单个组件抛出异常，那么它就会停止渲染整棵树。这种决策是为了提升安全性，同时也避免了状态不一致。

如果异常组件渲染失败，将它隔离，然后再渲染组件树的其他部分，这样做是不是更好呢？唯一可能实现这种效果的方式就是给渲染方法加上猴子补丁[①]，将它封装进 `try...catch` 语句中。这显然是很糟糕的做法，应该极力避免；不过某些情况下可以将它用于测试。

`react-component-errors` 库会给所有的组件方法加上猴子补丁，并封装到 `try...catch` 语句中，这样就不会导致整棵树渲染失败。

这种做法在性能与库的兼容性方面有一定缺陷，不过只要理解其中的风险，你就可以选用它。

先执行以下命令来安装这个库：

```
npm install --save react-component-errors
```

接着将它导入组件文件：

```
import wrapReactLifecycleMethods from 'react-component-errors'
```

然后用它装饰类：

```
@wrapReactLifecycleMethods
class MyComponents extends React.Component
```

这个库不仅可以避免单个组件异常导致整棵树渲染失败，还提供了设置自定义错误处理器的方式，并且可以在出现异常时获取有用的信息。

我们需要从包中导入 `config` 对象，如下所示：

```
import { config } from 'react-component-errors'
```

接着按以下方式设置自定义错误处理器：

```
config.errorHandler = errorReport => { ... }
```

定义为 `errorHandler` 的函数会接收错误报告，其中包含了重现并修复错误所需的信息。

除了原生错误对象，该报告还为我们提供了组件名与触发问题的函数名。此外，它还提供了组件接收的所有 props。这些信息足够用来编写测试、重现问题并快速修复。

值得强调的是，这个库所用的技巧应该予以避免，因为它可能会使应用出现某些问题。最重要的是，生产环境下应该禁用它。

① 原文为 monkey-patch，具体解释可查看 dongcia 在简书上的文章“猴子补丁”。——译者注

10.11 小结

本章介绍了测试的好处，以及可以用来覆盖 React 组件测试的框架。Jest 是一个**功能完整的工具**，而 Mocha 允许你**定制体验**。

TestUtils 允许你在浏览器外**渲染组件**。Enzyme 是一个强大的工具，可以在测试中**访问渲染输出的结果**。我们还探讨了如何用 mock 测试组件，以及如何编写预测代码。

我们还学到了快照测试能使组件输出的测试变得更简单，它的代码覆盖率工具也能帮助我们监控代码库的测试状况。

你需要牢记常见的复杂组件测试方案，当遇到高阶组件或者表单包含多层嵌套的元素时，它们就能派上用场了。

最后，我们学习了 React 开发者工具对测试的帮助，以及如何在 React 中实现错误处理。

第 11 章 需要避免的反模式

你已经在本书中学习了如何应用最佳实践来编写 React 应用。我们在第 1 章中回顾了基本概念以巩固扎实的理解，后面的章节介绍了一些更高级的技巧。

现在你应该能够开发可复用组件，使组件之间可以通信，并知道如何优化应用的组件树以达到最优性能。然而，开发人员也会犯错，本章将介绍使用 React 时应该避免的常见反模式。

了解常见错误有助于你避开它们，并且能够增进你对 React 的工作原理的理解，从而掌握 React 应用的开发方式。对于每个问题，我们都提供了示例来说明如何重现和解决。

本章包含如下内容。

- 用 prop 初始化状态导致意外结果的场景。
- 为何直接修改状态的做法不对，还会损伤性能。
- 如何选择正确的 `key` 属性来协助一致性比较器更好地工作。
- 为何在 DOM 元素上展开 props 对象不可取，以及有哪些替代做法。

11.1 用 prop 初始化状态

在本节中，我们将探究为何用父组件传来的 prop 初始化状态往往是一种反模式。这里用了**往往**一词，是因为接下来我们会看到，只要清楚理解了这种做法的问题所在，仍然可以决定使用它。

最好的学习方式之一就是阅读代码，因此我们将创建一个简单组件，用+按钮递增计数器。

以类的形式实现组件：

```
class Counter extends React.Component
```

它的 `constructor` 方法用 `count` 属性初始化状态，同时绑定事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
```

```
    count: props.count,
  }

  this.handleClick = this.handleClick.bind(this)
}
```

点击事件处理器的实现很简单：为当前计数值加 1，再将结果保存到状态。

```
handleClick() {
  this.setState({
    count: this.state.count + 1,
  })
}
```

最后，在 render 方法中声明由当前计数值和递增按钮组成的输出结构：

```
render() {
  return (
    <div>
      {this.state.count}
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

现在渲染该组件，将 count 属性设为 1：

```
<Counter count={1} />
```

组件按预期工作了：每次点击+按钮都会递增当前值。那么这有什么问题呢？

两个主要错误是：

- 我们违背了单一数据源原则；
- 传给组件的 count 属性发生变化时，状态不会相应地更新。

如果用 React 开发者工具审查 Counter 元素，那么我们会发现 Props 和 State 保存的值相同：

```
<Counter>
Props
  count: 1
State
  count: 1
```

这样就无法准确判断组件使用了哪个值并将其展示给用户了。

更糟糕的是，点击+按钮会导致两者不一样：

```
<Counter>
Props
  count: 1
State
  count: 2
```

11

此时我们可以推断，第二个值表示当前计数值，然而这很不明显，还可能导致意外行为或者向树下传递错误值。

第二个问题在于 React 如何创建并初始化类。创建组件时只会调用一次类的 constructor 方法。

Counter 组件读取了 count 属性的值，再将它保存在状态中。如果属性值在应用的生命周期内发生变化（比方说变成 10），那么 Counter 组件永远不会使用这个新值，因为它已经完成初始化了。这使得组件状态变得不连贯，这种情况很不理想，也很难调试。

如果我们真的想要用属性值初始化组件，并且可以肯定这些值未来不会改变呢？

这种情况下，最好阐明这种做法的用意，并为属性起一个能清楚表达含义的名称，如 initialCount。举例来说，按以下方式修改 Counter 组件的 constructor 方法：

```
constructor(props) {
  super(props)

  this.state = {
    count: props.initialCount,
  }

  this.handleClick = this.handleClick.bind(this)
}
```

然后按以下方式使用它：

```
<Counter initialCount={1} />
```

上述代码清晰地表明了父组件只能通过一种方式初始化计数器，以后为 initialCount 属性赋任何值都会被忽略。

11.2　修改状态

React 提供了非常清晰直观的 API 来修改组件的内部状态。setState 方法可以告诉库我们想要如何修改状态。一旦状态完成更新，React 会重新渲染组件，我们可以通过 this.state 属性访问新状态。就是这么简单。

然而有时我们会犯错，试图直接修改状态对象，这会对组件的连贯性和性能造成严重后果。

首先，如果不通过 setState 修改状态，则会出现两种糟糕的情况：

- 状态改变不会触发组件重渲染；
- 以后无论何时调用 setState，之前修改的状态都会渲染到页面上。

回到计数器示例中，修改点击事件处理器：

```
handleClick() {
  this.state.count++
}
```

此时点击+按钮不会影响浏览器中渲染的值，但如果用 React 开发者工具查看组件，会发现状态的值已经正确更新。这就打破了状态的连贯性，很显然，我们并不希望应用出现这种情况。

如果你这样做是因为失误，只要简单地用 setState API 来修复即可；但如果你是故意这样做，比如为了避免组件重渲染，最好重新思考一下组件的架构。

第 3 章中曾经提过，使用状态对象的原因之一就是保存 render 方法所需要的值。

直接修改状态引发的第二个问题是，无论组件的其他部分何时调用 setState，之前修改的状态都会渲染到页面上，这很出乎意料。

举例来说，继续修改 Counter 组件，为它添加以下按钮，点击后在状态中新建 foo 属性：

```
<button onClick={() => this.setState({ foo: 'bar' })}>
  Update
</button>
```

我们可以看到，点击+按钮不会产生任何视觉效果，但只要点击 Update 按钮，浏览器中的计数值就会突然跳变，显示出当前状态中隐藏的计数值。

这种不受控制的行为也是我们想要极力避免的。

最后非常重要的一点是，直接修改状态会严重影响性能。为了展示这种行为，我们将创建一个新组件，它和我们在第 9 章中学习 key 属性与 PureComponent 的用法时创建的列表组件很相似。

使用 PureComponent 时，直接修改状态值会带来负面影响。我们通过创建以下 List 组件来理解这个问题：

```
class List extends React.PureComponent
```

在 constructor 方法中初始化带有两个事项的列表，同时绑定事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
  }

  this.handleClick = this.handleClick.bind(this)
}
```

点击处理器非常简单——它往数组中推入新元素，再将数组设回状态（马上我们就会看到为何这种做法是错误的）：

```
handleClick() {
  this.state.items.push('baz')

  this.setState({
    items: this.state.items,
  })
}
```

最后，在 render 方法中显示列表的当前长度，并声明触发事件处理器的按钮：

```
render() {
  return (
    <div>
      {this.state.items.length}
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

查看上述代码，我们可能觉得没有任何问题；但如果在浏览器中运行组件，就会注意到点击+按钮不会更新列表的长度值。

此时只要用 React 开发者工具检查组件的状态，就会发现内部状态已经更新，但没有触发重渲染：

```
<List>
State
  items: Array[3]
    0: "foo"
    1: "bar"
    2: "baz"
```

出现这种不连贯体验的原因在于我们没有提供一个新数组，而是直接修改了原有数组。

实际上，往数组中推入新元素不会创建新的数组。PureComponent 通过检查组件的 prop 和状态是否改变来判断要不要更新，但本例前后两次传递了相同的数组。这种行为一开始会令人觉得违反直觉，特别是你不熟悉不可变数据结构时。

正确的做法应该是始终将新值赋给 state 属性，这个问题很好解决，只要按以下方式修改 List 组件的点击处理器即可：

```
handleClick() {
  this.setState({
    items: this.state.items.concat('baz'),
  })
}
```

数组的 concat 函数会将新元素拼接到原有数组上，然后返回新的数组。这样 PureComponent 就会发现状态中有了新数组，然后正确地重新渲染。

11.3　将数组索引作为 key

第 9 章中介绍性能和一致性比较器时，我们学习了如何用 key 属性帮助 React 判断更新 DOM 的最优路径。

因为 key 属性唯一标识了 DOM 中的某个元素，所以 React 用其判断元素是否为新的，以及组件属性和状态改变时是否要更新元素。

使用 key 始终是很好的做法，在没有用它的情况下，React 会在控制台给出警告（开发模型下）。然而 key 的使用没那么简单；用不同的值作为 key 有很大差别。实际上，某些情况下错误的 key 会引发无法预料的行为。本节将会介绍其中一个示例。

我们再来创建一个 List 组件：

```
class List extends React.PureComponent
```

在 constructor 方法中初始化 items 数组，并为组件绑定事件处理器：

```
constructor(props) {
  super(props)

  this.state = {
    items: ['foo', 'bar'],
  }

  this.handleClick = this.handleClick.bind(this)
}
```

点击事件处理器的实现与上一节中的稍有不同，因为这次我们要在列表顶部插入新元素：

```
handleClick() {
  const items = this.state.items.slice()
  items.unshift('baz')

  this.setState({
    items,
  })
}
```


最后，在 render 方法中声明列表，添加+按钮用于在列表顶部添加 baz：

```
render() {
  return (
```

```
    <div>
      <ul>
        {this.state.items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

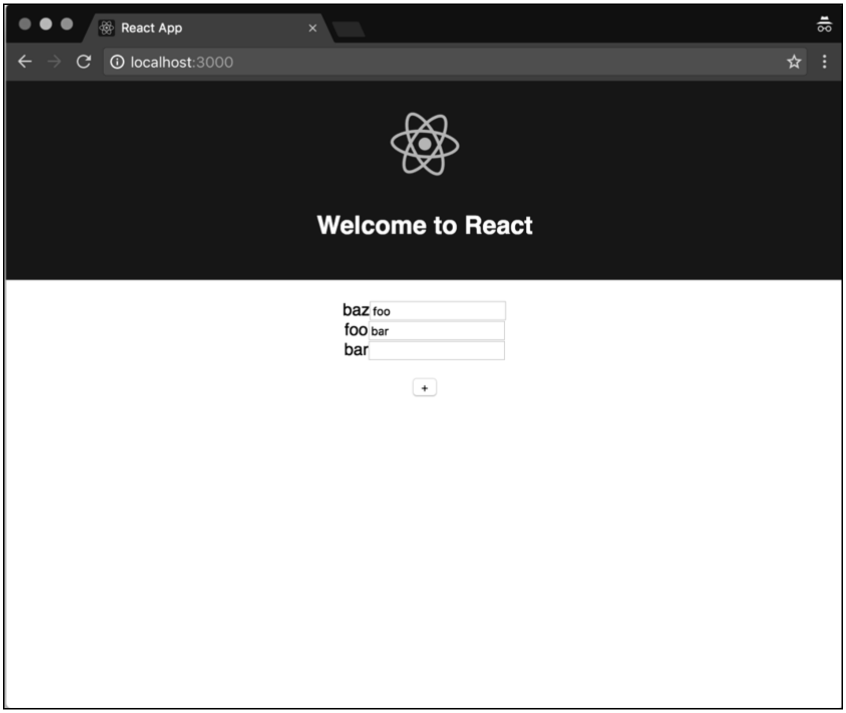
在浏览器内运行组件，结果一切正常：点击+按钮会在列表顶部插入新元素。不过我们接着进行一个试验。

按以下方式修改 `render` 方法，在每个列表项后添加一个输入框。输入框用于编辑各项内容，从而更容易理解问题所在：

```
render() {
  return (
    <div>
      <ul>
        {this.state.items.map((item, index) => (
          <li key={index}>
            {item}
            <input type="text" />
          </li>
        ))}
      </ul>
      <button onClick={this.handleClick}>+</button>
    </div>
  )
}
```

再次在浏览器内运行组件，将每项的值复制到输入框中，然后点击+按钮，此时就会出现意外行为。

如以下截图所示，原有的列表项向下移了一位，但输入框元素的位置保持不变，因此它们的值和列表项不再匹配了：



为了弄清并理解出现这个问题的原因，我们需要安装 Perf 插件并将它导入组件：

```
import Perf from 'react-addons-perf'
```

接着用两个生命周期钩子记录 React 操作 DOM 的信息，并打印到控制台中：

```
componentWillUpdate() {
  Perf.start()
}

componentDidUpdate() {
  Perf.stop()
  Perf.printOperations()
}
```

运行组件，并点击+按钮，检查控制台应该就能找到答案了。

我们发现 React 并没有在列表顶部插入新元素，而是交换两个已有元素的文本，并将最后一项当作新的插入列表底部。出现这种行为的原因是，我们将 map 函数的数组索引用作了 key 属性。

实际上，数组索引始终从 0 开始，即便在列表顶部推入了新元素，React 也会认为我们修改了两个已有元素的值，并在索引为 2 的位置添加了新元素。这种行为与没有用 key 属性时一模一样。

11

这种模式很常见，因为我们会认为提供任何形式的 key 都是最佳做法，但实际并非如此。key 的值必须唯一且稳定，它只能标识唯一一项数据。

要想解决这个问题，可以用各项的值作为 key，前提是它们在整个列表中不会重复，或者也可以创建唯一标识符。

11.4　在 DOM 元素上展开 props 对象

Dan Abramov 最近将一种常见做法称作反模式，在 React 应用中这样做会触发控制台中的警告。

这个技巧在社区中广为使用，我个人也在真实项目中多次见到。我们常常会在元素上展开 props 对象，以避免手动编写每个属性，如下所示：

```
<Component {...props} />
```

这种写法可以正常运行，它会被 Babel 转译为以下代码：

```
React.createElement(Component, props);
```

然而，当在 DOM 元素上展开 props 对象时，就会有添加未知 HTML 属性的风险，这是很糟糕的做法。

问题不仅和展开操作符有关，逐个展开非标准属性会导致相同的问题和警告。因为展开操作符**隐藏**了我们所要展开的属性，所以很难判断具体向元素传递了什么。

渲染以下组件，这个基本操作就会触发控制台中的警告：

```
const Spread = () =>  <div foo="bar" />
```

警告信息如下所示：

```
Unknown prop `foo` on <div> tag. Remove this prop from the element
(<div>标签包含未知的 foo 属性，请从元素上移除它)
```

因为 foo 属性对于 div 元素来说是无效的。

如上文所说，很容易在本例中找到传递的属性并移除，如果用了展开操作符，如下所示：

```
const Spread = props => <div {...props} />
```

我们无法控制父组件会传来哪些属性。

如果按以下方式使用组件：

```
<Spread className="foo" />
```

没有任何问题。

但如果按以下方式的话：

```
<Spread foo="bar" className="baz" />
```

React 就会给出警告，因为我们在 DOM 元素上设置了非标准属性。

解决这个问题的其中一种方法就是创建一个名为 domProps 的属性，可以在组件上安全地展开它，因为我们显式声明了它包含有效的 DOM 属性。

例如，我们可以按以下方式修改 Spread 组件：

```
const Spread = props => <div {...props.domProps} />
```

用法如下所示：

```
<Spread foo="bar" domProps={{ className: 'baz' }} />
```

正如我们多次见到的那样，在 React 中采取显式做法始终是很好的实践。

11.5　小结

了解所有的最佳实践永远不会错，但知道反模式有时能帮助我们避免走错方向。最重要的是，学习某些技巧被看作糟糕实践的原因有助于我们理解 React 的工作原理，并掌握它的高效用法。

本章介绍了四种不利于 Web 应用的性能和行为的组件用法。

对每种用法，我们用示例重现了问题，并提供了具体的修复方案。

我们学习了为何用属性初始化状态会导致两者不一致，还发现了直接修改状态会损伤性能。另外，我们还了解到错误的 key 属性会给一致性比较算法带来负面效果。最后，我们学习了在 DOM 元素上展开非标准属性被视为反模式的原因。

第 12 章 未来的行动

React 是近几年出现的最令人惊叹的库之一，不仅因为它拥有众多强大的特性，更因为一个生态系统围绕它发展起来了。

追随 React 社区十分激动人心，并且能启发我们的灵感：每天都可以学习和使用新的项目和工具。不仅如此，你还可以参加很多会议和线下聚会，与其他人交流，并结识新朋友。此外，还可以阅读大量博文，以提升自己的技能并学习更多知识。总之，你能找到很多途径成为更好的开发人员。

React 与其生态系统十分鼓励最佳实践和开源，这对开发人员未来的职业生涯非常有利。

本章包含如下内容。

- 如何通过提交问题和发起 pull request 为 React 库做贡献。
- 为何回馈社区以及分享自己的代码很重要。
- 发布开源代码时需要牢记的重点。
- 如何发布 `npm` 包和使用语义化版本号。

12.1 为 React 做贡献

使用 React 一段时间后，人们往往会想着为这个库做些贡献。React 是开源项目，这意味着它的源代码是公开的，签署了**贡献者许可协议**（contributor license agreement，CLA）的任何人都能参与修复 bug、编写文档，甚至添加新特性。

可以搜索“Contributing to Facebook Projects”来阅读完整的 CLA 条款。

举例来说，假设用 React 开发应用时发现了 bug，应该怎么做呢？首先，也是最重要的一点是，创建一个简单的示例来重现问题。这一步可以用 React 团队提供的现成的 JSFiddle 模板：

https://jsfiddle.net/reactjs/69z2wepo/

这样做主要有以下两个好处：

- 可以帮你百分之百地确定这是 React 的 bug，而不是你自己的应用代码的问题；
- 有助于 React 团队快速理解问题，无须深入研究你的应用代码，从而可以加快修复过程。

JSFiddle 模板用的是最新版的 React，这一点很重要，因为如果你发现的是旧版 React 的 bug，那么最新版本可能已经修复了这个 bug。反之亦然，如果你发现最新版有问题，而旧版没有，那么修复该 bug 的优先级就会很高，因为它可能会影响大量用户。

准备好展示问题的示例后，就可以在 GitHub 上提交问题：

https://github.com/facebook/react/issues/new

你会看到，提交问题的页面已经预填了一些引导提示，其中一条是提交示例的最低要求。其他的内容有助于你解释问题，并描述当前和期望的行为。

参与或贡献代码前，最好阅读一下 Facebook 行为准则（https://code.facebook.com/codeofconduct）。该文档列出了所有社区成员共同期望并且每个人都应该遵守的良好行为。

提交问题后，需要等待某个核心贡献者审阅，然后他会告诉你 React 团队决定如何处理这个 bug。根据问题的严重程度，他们可能会自己修复，也可能请求你来修复。

在第二种情况下，你可以 fork 这个 GitHub 仓库，然后编写代码解决问题。请遵循代码风格指南，并为补丁编写全面的测试。还有一点很关键：确保新代码能通过现有的所有测试，避免向代码库引入新的错误。

问题修复并通过所有测试后，就可以发起 pull request，然后等待核心团队成员审核代码。他们可能会决定合并，也可能要求你进行一些修改。

如果没有发现 bug，但又想为这个项目做一些贡献，那么可以在 GitHub（https://github.com/facebook/react/labels/good%20first%20issue）上查看带有 good first issue 标签的问题。

这是开始贡献的一种绝佳方式，其妙处在于，React 团队为每个贡献者（尤其是新人）都提供了参与项目的机会。

如果发现某个带有 good first issue 标签的问题还没有被其他人认领，那么你可以在它的页面上评论，表明你有兴趣修复它。随后，核心成员就会联系你。在开始编写代码前，请和他们讨论你的思路以及解决方式，以免多次重写代码。

改进 React 的另一种方式是添加新特性。值得一提的是，React 团队有自己的计划，核心成员负责设计并选择主要特性。

如果对 React 的发展路线感兴趣，那么你可以从 GitHub（https://github.com/facebook/react/

issues?q=is:open+is:issue+label:%22big+picture%22）上带有 big picture 标签的问题中了解部分信息。

也就是说，如果对 React 应该新增的特性有好的建议，那么你可以立刻提交问题，并与 React 团队交流。在那之前，最好不要花时间编写代码和发起 pull request，因为你构想的特性可能不符合 React 团队的计划，也可能与他们正在开发的其他功能有冲突。

12.2　分发代码

为 React 生态系统做贡献并不只是向 React 仓库提交代码。要想回馈社区和帮助其他开发人员，你可以开发软件包、撰写博文、在 Stack Overflow 回答问题，还可以做很多其他事情。

举例来说，假如你开发了一个可以解决复杂问题的 React 组件，你认为其他开发人员也可以从中受益，他们可以直接使用这个组件，而不需要花时间研究自己的方案。最佳做法是将它发布到 GitHub 上，允许每个人查看和使用。不过，在 GitHub 上发布代码只是整个大流程中的一小步，它也伴随着一些责任。因此，你心里应该清楚地认识到这种选择背后的理由。

这样做的动力之一是，你可能想要通过贡献代码来提升自己的开发技能。一方面，共享代码需要你遵循最佳实践，编写更好的代码；另一方面，代码要接受其他开发人员的反馈和评论。这是一个很好的机会，你可以接受建议并改进代码，使它更完美。

除了与代码本身相关的建议以外，将代码发布到 GitHub 还能从他人的想法中得到很多启发。你可能只考虑用自己的组件解决单一问题，而其他开发人员或许会以不同的方式来使用它，从而发现新的用途。此外，他们可能需要新的特性，并可以帮助你实现，这样包括你在内的每个人都能从中受益。合作开发软件可以极大地提升你的技能、改进你的软件包，这也是我对开源充满信心的原因所在。

开源还可以为你提供与全世界聪明热情的开发人员交流的绝佳机会。与拥有不同背景和技能的新伙伴一起紧密合作是开放思维和提升自己的最佳方式之一。

共享代码也带来了一些责任，而且可能占据你大量时间。实际上，开放代码让他人使用后，你就要负责维护它。

维护代码仓库需要付出，随着它变得越来越流行，用户会越来越多，疑问和问题也会大大增多。举例来说，开发人员会遇到 bug 并提交问题，此时你就要全部浏览并尝试重现问题。如果问题确实存在，那么你就需要编写补丁并发布新的版本。你还会收到其他开发人员发起的 pull request，其中的代码可能又长又复杂。即便如此，你仍然要审查它们。

如果你决定邀请他人共同维护项目，并帮助你处理问题和 pull request，那么你就要在合作过程中分享自己的观点，并和他人共同决策。牢记这一点，接着我们来了解一些优秀实践，它们可

以帮助你更好地维护代码仓库，同时避免一些常见的陷阱。

首先，如果想要发布 React 组件，那么你需要编写全面的测试集。对于多人参与编写的公共代码来说，测试的帮助很大，理由如下：

- 测试使代码更稳健；
- 测试能帮助其他开发人员理解代码的功能；
- 测试有助于发现新增的代码缺陷；
- 测试让参与编写代码的其他开发人员更有信心。

第二点，添加描述组件的 README 文件，其中包括使用示例、API 文档以及可用的 prop。

该文件对软件包的用户很有帮助，而且也能避免人们提交问题询问库的工作原理和用法。

另外，在仓库中添加 LICENSE 文件也很重要，它可以提醒人们如何恰当地使用你的代码。GitHub 提供了大量现成的模板供你选择。

你应该尽量减小软件包并少用依赖。在决定是否使用某一个库时，开发人员往往会仔细考虑它的大小。记住，大软件包会给性能造成负面影响。

不仅如此，依赖太多第三方库会带来问题，因为其中的某些库可能有 bug 或者不再有人维护。

在分享 React 组件时，样式方面的决策非常麻烦。分享 JavaScript 代码十分简单，而一旦涉及 CSS，就没那么容易了。其实，你可以选择许多不同的方式来提供样式：可以在包内添加 CSS 文件，也可以使用行内样式。最重要的一点是，CSS 作用于全局，导入组件后的一般类名可能会与项目中已有的类名发生冲突。

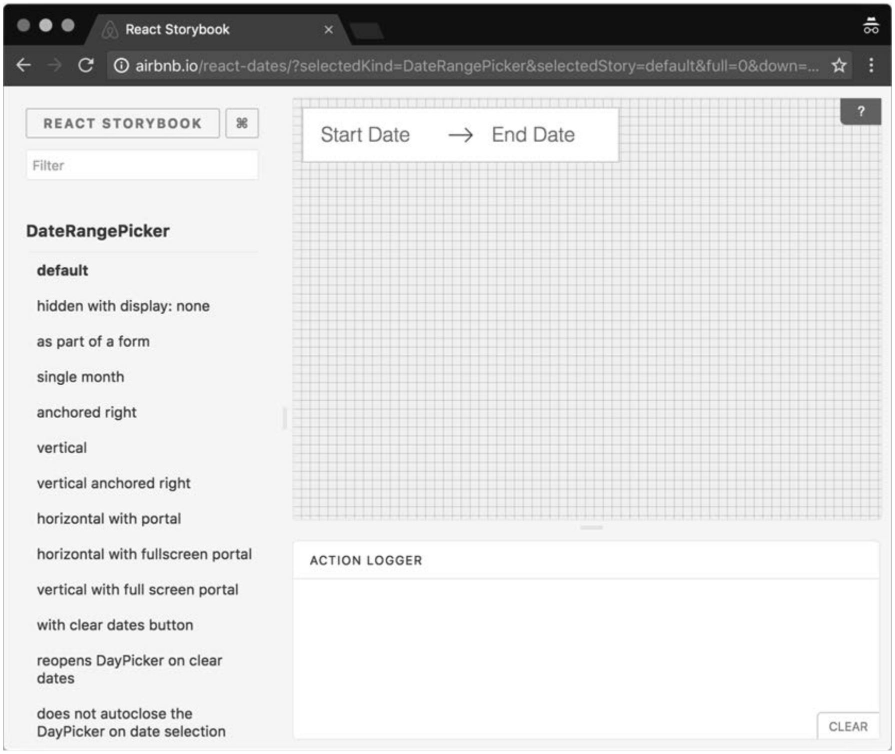
最佳选择是尽量少提供样式，允许用户自由配置组件。这样一来，开发人员就更愿意使用它，因为这能和他们自己的技术方案相匹配。

为了向他人展示你的组件可以非常灵活地进行配置，可以在仓库中添加一些示例，以便他人更容易理解组件的工作原理，以及它接受哪些 prop。你也可以用这些示例测试新版组件，并查看是否有意外的破坏性改动。

正如我们在第 3 章中曾见到的，React Storybook 这类工具可以创建现成的风格指南，不但维护起来更容易，而且允许用户更方便地浏览和使用软件包。

Airbnb 的 `react-dates` 组件是一个配置化程度很高的完美示例，它用 Storybook 展示了组件的所有状态。你可以将它的代码仓库作为绝佳的范例，以学习如何在 GitHub 上发布 React 组件。

Airbnb 的开发人员用 Storybook 展示了组件的不同选项，如下所示：



最后一点很重要：你可能不只是想要分享代码，还想要发布软件包。npm 是 JavaScript 最流行的包管理器，本书就用它安装了各种包和依赖。

我们将在下一节中介绍，用 npm 发布新软件包其实很简单。

除了 npm，有些开发人员可能需要将你的组件用作全局依赖，而且不想通过包管理器来使用。

第 1 章中曾经提过，React 的使用很简单，只要添加一个<script>标签，将其路径指向 https://unpkg.com 即可。为你的用户提供相同的安装选项很重要。

因此，为了提供全局版本的软件包，你还需要构建符合**通用模块定义**的版本。用 Webpack 很容易做到这一点：在配置文件的输出部分设置 libraryTarget 属性即可。

12.3　发布 npm 包

为开发人员提供软件包的最流行做法就是将它发布到 npm，即 Node.js 的包管理器。

本书中的所有示例都用到它了，你也看到了，用它安装包非常简单：只要执行 npm install <包名>即可。你可能还不了解的是，用它发布软件包也非常简单。

首先，进入空目录，并在终端中执行以下命令：

```
npm init
```

此时会新建 package.json 文件，并显示几个问题。第一个问题会问你包名是什么，它默认是文件夹名称，然后会问版本号。这些信息最为重要，因为用户安装并使用软件包时要靠包名来引用；版本信息可以帮你安全地发布新版，避免与他人的代码发生冲突。

版本号由三个以点号分隔的数字组成，它们都有各自的含义。版本号中的最后一个数字表示补丁，当库的新版包含 bug 修复补丁时，发布到 npm 上需要增加该数字。

中间的数字表示次要版本，库新增特性时需要修改它，并且这些特性不能破坏已有的 API。

最后，第一个数字表示主版本，公开发布的版本包括重大改动时应该增加该数字。

这种版本命名方式称为**语义化版本控制**，它是一种良好的实践，能让用户在更新软件包时更放心。

包的第一个版本号通常是 0.1.0。

要想发布 npm 包，必须有 npm 账户，在控制台中执行以下命令即可创建账户：

```
npm adduser $username
```

$username 表示你选择的用户名。

有了账户后，就可以执行以下命令：

```
npm publish
```

这时，npm 就会用 package.json 中指定的包名和版本号注册一个入口。

任何时候要想修改库并发布新版，只需要执行以下命令：

```
npm version $type
```

$type 可以是补丁版本、次要版本或者主版本。这条命令会自动修改 package.json 中的版本号，如果当前文件夹处于版本控制之下，它还会创建 commit 并打上标签。

版本号更新后，只要再次执行 npm publish，用户就可以下载新版了。

12.4 小结

在 React 世界之旅的最后一站，我们学习了成就 React 重要地位的几个方面：它的社区与生态系统，以及如何才能参与其中。

你学习了发现 React 的 bug 时应该如何提交问题，以及让库的核心开发人员更方便地修复问题需要哪些步骤。现在，你还知道了开源代码的最佳实践，以及伴随而来的好处与责任。

最后，你学到了在 `npm` 上发布软件包非常简单，并学习了如何选择正确的版本号以避免破坏他人的代码。

微信连接

回复“React”查看相关书单

微博连接

关注@图灵教育 每日分享IT好书

QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

本书介绍如何构建更加灵活、运行流畅、易于维护的应用，让开发人员在不降低质量的情况下极大地提升工作流的速度。读者将首先了解React的内部原理，开发能够在整个应用中复用的组件，搭建应用架构，创建真正可用的表单；随后会为React组件编写样式并优化组件，编写测试代码；最后还会学到如何为React及其生态系统做贡献。

- 编写整洁、易维护的代码
- 应用技巧创建可复用的组件
- 在浏览器和节点中有效运用React
- 根据应用的需要选择美化方案
- 使用服务端渲染提升应用加载速度
- 通过优化组件来构建高性能应用

[PACKT]
PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/软件开发
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-48875-6
ISBN 978-7-115-48875-6
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks